AD-A053 328 MASSACHUSETTS INST OF TECH CAMBRIDGE LAB FOR COMPUTE--ETC F/G 9/4
ACTOR SYSTEMS FOR REAL-TIME COMPUTATION.(U)
MAR 78 H G BAKER N00014-75-C-0522
UNCLASSIFIED MIT/LCS/TR-197 NL

1 OF 2
AD A053328

LABORATORY FOR COMPUTER SCIENCE

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

(12)

MIT/LCS/TR-197

ACTOR SYSTEMS FOR REAL-TIME COMPUTATION

Henry G. Baker, Jr.

This research was supported by the Office of Naval Research under Contract No. N00014-75-C-0522

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: MIT/LCS/TR-197	2. GOVT ACCESSION NO.
3. RECIPIENT'S CATALOG NUMBER: Doctoral thesis	
4. TITLE *(and Subtitle)*: Actor Systems for Real-Time Computation	5. TYPE OF REPORT & PERIOD COVERED: Ph.D. Thesis, Feb.14,1978
	6. PERFORMING ORG. REPORT NUMBER: MIT/LCS/TR-197
7. AUTHOR(s): Henry G. Baker, Jr.	8. CONTRACT OR GRANT NUMBER(s): N00014-75-C-0522
9. PERFORMING ORGANIZATION NAME AND ADDRESS: MIT/Laboratory for Computer Science, 545 Technology Square, Cambridge, Ma 02139	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS: Office of Naval Research, Department of the Navy, Information Systems Program, Arlington, Va 22217	12. REPORT DATE: March 1978
	13. NUMBER OF PAGES: 147 (12) 148 pp.
14. MONITORING AGENCY NAME & ADDRESS *(if different from Controlling Office)*	15. SECURITY CLASS. *(of this report)*: Unclassified
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT *(of this Report)*

Approved for public release; distribution unlimited

17. DISTRIBUTION STATEMENT *(of the abstract entered in Block 20, if different from Report)*

18. SUPPLEMENTARY NOTES

19. KEY WORDS *(Continue on reverse side if necessary and identify by block number)*

Real-time	Garbage collection
Parallel	List memory
Message-passing	Continuations
Semantics of Parallelism	Models for distributed computation
Storage management	Partial orders of events

20. ABSTRACT *(Continue on reverse side if necessary and identify by block number)*

Actor theory was invented by Hewitt and collaborators as a synthesis of many of the ideas from the high-level languages LISP, GEDANKEN, SMALLTALK, SIMULA-67 and others. Actor theory consists of a group of active objects called ACTORS, which communicate by passing messages to one another. This thesis explores several problems associated with implementing Actor theory as a basis for computer system design. First, we give a firmer foundation to the theory by setting forth axioms which must be satisfied by any physically realizable

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

20. message-passing system. We then give an operational semantics for this theory by exhibiting an interpreter which is a concrete model for the theory. Thirdly, we explore the implementation questions of mapping this conceptual system onto current hardware in such a way that simple primitive operations all take a (small) bounded amount of time. In particular, the issues of storage and processor management are investigated and a real-time incremental garbage system for both is exhibited and analyzed.

(12)

MIT/LCS/TR-197

Actor Systems for Real-Time Computation

by

Henry Givens Baker, Jr.

March 1978

This research was supported by the Office of Naval Research under contract number N00014-75-C-0522.

Massachusetts Institute of Technology
Laboratory for Computer Science

Cambridge Massachusetts 02139

Actor Systems for Real-Time Computation

by

Henry Givens Baker, Jr.

Submitted to the Department of Electrical Engineering and Computer Science
on February 14, 1978 in partial fulfillment of the requirements
for the Degree of Doctor of Philosophy.

ABSTRACT

Actor theory was invented by Hewitt and collaborators as a synthesis of many of the ideas from the high-level languages LISP, GEDANKEN, SMALLTALK, SIMULA-67, and others. Actor theory consists of a group of active objects called *Actors*, which communicate by passing messages to one another.

This thesis explores several problems associated with implementing Actor theory as a basis for computer systems design. First, we give a firmer foundation to the theory by setting forth axioms which must be satisfied by any physically realizable message-passing system. We then give an operational semantics for this theory by exhibiting an interpreter which is a concrete model for the theory. Thirdly, we explore the implementation questions of mapping this conceptual system onto current hardware in such a way that simple primitive operations all take a (small) bounded amount of time. In particular, the issues of storage and processor management are investigated and a real-time incremental garbage collection system for both is exhibited and analyzed.

Thesis Supervisor:
Carl Hewitt, Associate Pr ssor of Electrical Engineering and Computer Science

Acknowledgments

I would like to thank my thesis supervisor, Professor Carl Hewitt, for his support and encouragement. Carl's thirst for simplicity and his wide ranging interests provided an excellent sounding board for my many half-baked ideas. Thanks also go to my readers, Professor Barbara Liskov and Professor Stephen Ward, through whom I was able to gain a different perspective on many topics. I enjoyed many fascinating bull sessions with Professors Vaughan Pratt and Albert Meyer, and Dr. Peter Jessel provided me with much personal and professional support. In spite of MIT, the graduate students and staff are superb; many thanks to Richard Greenblatt, Tom Knight, and Guy Steele and many others for their help and for making some of the best engineered computer systems in the world. Peter Bishop, John DeTreville, Bert Halstead, Al Mok, and Eliot Moss also made suggestions for improvement. Finally, my warmest thanks and gratitude go to my wife, Carolyn, a doctor in her own right, for her love, support, and patience during these past seven years.

This thesis was typed into the MIT A.I. Lab. PDP-10 "TECO" program (magnificently maintained by Richard Stallman), using "DOC", a set of TECO macros written by Vaughan Pratt. It was printed on the A.I. Lab. Xerographic Printer, after being formatted by Alan Snyder's document compiler "R".

This research was supported by the Advanced Research Projects Agency of the Department of Defense and was monitored by the Office of Naval Research under contract number N00014-75-C-0522.

CONTENTS

FIGURES

1. Background

1.1 Introduction

Hardware is cheap. Software is dear. These are the cliches of the computer industry in the 1970's. The proposals in this thesis will hopefully trade off slightly higher costs in hardware for greatly increased productivity in software. Systems savings should be significant for several reasons. Better software productivity means less time in development and more in the marketplace. Since this increase in productivity results in part from a more direct mapping of ideas into programs, costs of debugging and maintenance should be less. Because the underlying computational model (Actors) is closer to the conceptual objects (abstract data types) of the program, fewer approximations and compromises have to be made, resulting in more robust programs. Greater software productivity means that software costs are less, allowing for more specialized programs since decreased software costs can be spread over fewer installations. In many cases, any increased costs in the hardware will be offset by the increased efficiency in the software--efficiency gained by exploiting global instead of peephole optimizations.

Ample evidence exists for the efficacy of this kind of tradeoff. The ubiquity of the *operational amplifier* instead of the simpler (to make), cheaper, and "more efficient" transistor is a result of this "mind-over-matter" tradeoff. Even though no simple, elegant operational amplifier gate exists, it is a tribute to the mathematical elegance of this device and its ease of use in design that integrated circuits having large numbers of circuit elements are being made to simulate operational amplifiers. The existence of the "op amp" allows this single concept to replace a wide variety of real devices; a conceptual economy that simplifies synthesis procedures, analyses, and inventories. Thus, the "op amp" is a paradigm for what can be achieved when total system design costs are fully accounted for.

Of course, now that op amps are the accepted standard, their manufacturers will diligently search for new gates and devices to implement this mathematical model more cheaply. Perhaps as a result of this research a single element op amp will appear which is

simpler and cheaper than a transistor. Technology would then catch up with theory.

This thesis argues that current Von Neumann computer architecture is as ill-suited to computer systems design as the transistor is to electronic circuit design; it is reliable and cheap, but a poor match to the problem domain. The system designer would like to think about high level objects like queues, data-bases, I/O streams, program modules and operations on those objects like "insert", "print", "delete". Current day computers offer only bits, character strings, and numbers, and the size of the objects that can be conveniently operated on is restricted by the fact that these objects must be examined or moved as a unit, in their entirety. Thus, one can never "get off the ground", so to speak, because the computer is destined to work only with those trivial entities.

Actor theory was invented by Carl Hewitt and collaborators [49,85,43,44,51,50,52] in response to these problems as a synthesis of many other ideas about abstract data types and control structures [27,15,26,39]. In this theory, actors and messages are the only two types of objects. Actor systems exhibit behavior through actors sending messages to other actors, which in turn send more messages. Actors can be created in the course of a computation, and their names can be communicated in messages without sending the actors themselves in the messages. Hence, elements of high-level data types can be modelled quite effectively as actors which receive messages indicating the high level operations that they should perform--perhaps on themselves. The actor model also allows for concurrent processing because many actors can be receiving and sending messages independently of one another. Thus, in addition to the actor theory being universal, which by itself is no great prize, it matches very well (some of) our intuitions about how physical, computational, and conceptual systems work. Thus, we propose to make the actor the "op amp" of computation.

1.2 Distributed Computing

In recent years, there has been a shift from the centralized serial computing system to the distributed parallel computing network. The large, general-purpose computer of the IBM System/360 style is being replaced by numerous more dedicated mini and

microcomputers connected together by phone lines, satellite links, Ethernets, and the like.

There are many driving forces behind this shift. Since information is often produced at a different geographic location from where it is consumed, it must be transmitted. With the costs of both digital communication and the smallest viable computer dropping, it is becoming easier and more economical to digitize and edit the data at its source, so that only the editted data need be transmitted. Response time for editting these trivial requests can drop dramatically when there is a computer on-site. System reliability may also be improved, because the loss of a particular node or link in the network need not completely shut down the system; i.e. it becomes fail-soft.

But centralized computing facilities are also undergoing change. The costs of CPU's have continued to drop until they are only a small sliver of the computer budget. The cost per bit of memory has also dropped at the same rate, but instead of systems now costing less, the size of memories has grown to keep the overall system cost constant. The $64,000 question is "How to take better advantage of all this memory to increase throughput?" Faster CPU's are not the answer since they require more expensive high-bandwidth memories, and memory cost is already the largest single cost in the system. Neither are more CPUs the answer, because in the current "shared memory" paradigm they must still be connected to the same memory, and the memory bus becomes the bottleneck.

The answer that is becoming increasingly clear is to associate some computing power closely with each chunk of memory and replace the "shared-memory" paradigm with the "society of computers" paradigm. The hope is that the computation of each CPU will be local enough to reduce the communication bandwidth required between memory chunks by at least an order of magnitude. This arrangement allows for better utilization of the scarce resource--memory--than any of the other alternatives.

The cost of the smallest viable computer going down, and its cost is going down faster than its computing power. In other words, computing power has gotten cheaper in absolute terms. However, the optimum MIPS/$ does not come at the fast end of the scale, with sub-nanosecond gates, but in the relatively slow range achieved by microprocessors on a single chip. Therefore, massive parallelism may allow performance from these chips equal

to the performance of large serial computers such as the Cray-1. Mechanisms and methods must be found which use this additional computing power to produce the answer to a given computing request faster.

This thesis investigates several problems which systems with large numbers of independent computing elements must face. First, there are currently no good conceptual models for thinking about these systems. Many people are working on this problem and offer models with widely varying degrees of generality and efficiency; it is not yet clear where the tradeoff lies between these two conflicting goals. Second, while we would like to make programs run faster on cheaper hardware, it has become clear that for a vast number of situations, the cost of programming--especially testing and verification--is the limiting factor. Therefore, we would like to move to a message-passing paradigm, but not give up the hard-won gains in programmer productivity from advances in the serial computing art.

We attack the problem of the lack of good conceptual models through advances in the theory of Actors. We put this theory on a firmer foundation through axioms which specify the behaviors of actor computations. We also present a constructive model for these axioms which can be used as a gedanken interpreter for actor computations. We argue that event-based correctness proofs often avoid the exponential blowup of the classical "consider all shuffles" approach.

We attack the problem of programmer productivity on this new hardware configuration on several fronts. It has become clear that one way of increasing a programmer's productivity is to give her the tools to think about her problem in a high level way. In other words, instead of programming in terms of bits, bytes, and blocks, the programmer should manipulate pictures, accounts, warehouse inventories, etc. as "first-class" data types, and leave it to the compiler and/or interpreter to make it all work efficiently.

Actor theory provides a clear conceptual model for these types of programmer-defined data types since it unifies the concepts of *program* and *data*. An *account* actor, besides containing the data necessary to describe the account, also has a program called a *script* which allows the account to respond to high level requests such as: "what is your 60 day balance?" or "credit account with payment of $20.95". Once these high level data types have

been defined, most programs wither away to a few lines of code which send messages off to these now-active data objects to perform the real work.

However, the trouble with systems which provide user-defined data types has been that either the programmer has to know far more about the details of the implementation than is healthy, or she has to put up with gross inefficiency and perhaps intolerable run-time delays resulting from the automatic management of these objects.

This thesis helps solve one of the biggest problems in systems which take responsibility for these user-defined data types--*the management of storage.* This is significant because as we have argued it is not CPU time, but access to storage which is the limiting factor in computer performance. Since currently programmers are forced to explicitly manage this scarce resource without much help from either hardware or software, violations of storage management policies are hard to detect and cause havoc when they occur [16]. We claim that a system which uniformly and efficiently managed storage would increase programmer productivity manifold, especially in the program debugging stage, and would even tend to do a better job at it than the programmer could.

There are two reasons for this. First, even though for any *particular* task the programmer can probably do a better job of storage management than the system, the many small domains of explicit storage management which result can lead to an overall reduction in total system efficiency, because storage can not be easily reallocated when some domain becomes full. For example, a stack overflows when there is still plenty of room left in the hash table. This is the classical fragmentation problem--"storage, storage everywhere, but not a byte to munch!" A uniform, global strategy would allow the system to allocate storage only where and when it is needed.

The other reason why a uniform system can do a better job of storage management is that while programmers *can* do a better job, they usually *don't*, because they are pressed for production, and it is not worth their valuable time to optimize storage utilization.[1]

By freeing the programmer from worrying about the management of storage (allocation and freeing), it leaves her more time to worry about more important questions, such as the correctness of the program or the scheduling of various parts of the program to

achieve better response time. Time scheduling [24] is apparently a much harder problem than storage management, or else it is not so well understood; hence it is more important for the programmer to worry about the management of time than the management of storage.

(After all, human beings have many automatic systems to manage their fluids, their energy resources, their ion balances, etc., but time scheduling for humans is still a very high level function. In this analogy, automatic storage management functions less like the brain than the kidney, which continually reprocesses the bodily fluids to maintain the right environment for the more important functions.)

1.3 Actors in Hardware

A revolution is currently taking place in the computer industry. For the first time, more CPU cycles are available than we know what to do with. This is due to the availability of microprocessors on a single chip that can be turned out almost as fast and cheap as copies from a Xerox machine. Some of these single-chip computers come complete with on-board ROM (for program storage), RAM (for data storage), and I/O capability, requiring only a power supply and some I/O devices for operation.

Yet most computation remains expensive, far more expensive than these cheap micro-computers would lead us to believe. This is because system design and programming costs have remained high, or even increased, with the availability of these cheap computers.

There are many reasons for this. First, many of the lessons learned at great cost on mainframe computers are being re-learned at the micro-level; e.g. high-level languages can cut the cost of programming and maintaining large systems, yet micro-computer system developers continue to use "assembly" languages, many without even crude macro facilities.

---

1. One will notice that exactly the same reasons hold for using a dynamic uniform paging algorithm instead of manual overlays to manage programs that do not fit into primary memory. The paging system cannot perform as well on any particular stretch of code, but it is uniformly good on almost all of the code because it has access to dynamic run-time information. Therefore in most situations, the paging system does much better than manual overlays.

Second, program development requires a quite different environment than the running environment of the finished product; it requires editors, debuggers, sophisticated file systems, compilers for documentation, etc. Again, micro-computer development limps along using micro-computers themselves for editing, compiling, etc., tasks which may be inappropriate for these devices.

Third, old habits die hard. Faced with the prospect of cheap memory and cheap CPU cycles, programmers continue to apply techniques to conserve memory and multiplex CPU's which are inappropriate for the current hardware/software cost ratios. Time, not storage or CPU cycles, has always been of the essence, both in development and in product performance, but systems are continually evaluated in terms of their hardware cost only, not the software and opportunity costs which dominate.

Fourth, systems designers have missed seeing the forest for the trees. The real bottlenecks in computing are in communicating information between modules and not in the internal operation of any of the individual modules. Most CPU's spend a considerable fraction of their time waiting for I/O devices such as disks or in serially searching some small region in fast memory while the rest of the fast, and very expensive, memory sits idle. Yet the answer is not in simply adding more CPU's, because the bottleneck is still in the communication link between the CPU and memory, not in the CPU.

What is needed is some way of designing a system with a larger ratio of CPU cost to memory cost so that a larger percentage of the memory is being utilized most of the time.

The answer given by this thesis is not to design systems using CPU's and RAM's as separated components, with caches, sophisticated instruction sets, and clever algorithms to get back some efficiency, but to design systems with large numbers of very simple *actors*, each of which combines both a CPU and a small amount of RAM. These actors communicate not by interrogating a shared memory but by sending messages to one another. The best mechanism to transmit and deliver these messages has yet to be developed, but a full "telephone exchange" network like a Batcher sorting net [10] looks promising.

The speed and power of actor systems depends not upon the speed and power of the individual actors, which might be very dumb and slow, but on the massive parallelism of thousands and millions of these devices working in concert. The clever algorithms which have been developed for searching on serial computers to minimize the bandwidth required between the CPU and RAM are not needed in an actor system where hundreds of CPU's can be searching their local memories simultaneously. Even if each CPU is slow, and uses a naive search method, the search cannot take very long because each local memory is small.

We conceive of chips in the near future on which the large majority of the area is taken up by memory, and a CPU squeezed in around the margins. A few more years will see large arrays of CPUs all on a single chip, giving the power of the ILLIAC-IV [9] but with a good deal more flexibility.

The key to the current micro-computer revolution was the realization that one did not need all the complexity of the big computer instruction sets to build a Turing universal device that was still fast enough for many simple applications. Making the CPU simple allowed it to fit on a single chip.

Making the CPU even simpler is the key to the next revolution. Rather than trying to get a lot of power from one sophisticated CPU working alone, we plan to get that power by the joint effort of many simple devices working together. Each CPU should be universal,[2] but it must also be as simple as possible so that many will fit on a chip. The CPU does not need a clever instruction set, because it does not have to be speed or storage optimized; e.g. ten to twenty instructions are sufficient to perform the simple tasks that are required.

The content-addressable memory fad of the 1960's had the right idea--increase the memory bandwidth--but its advocates were slightly misguided. They hoped that by adding a little logic--a comparator, flag bits, etc. to a memory cell, the proper tradeoff would be

---

2. Each element of an array of parallel processes need not be universal for the array to be universal, viz. Conway's LIFE game [38] or Hennie's iterative arrays [47]. However, universality can be achieved with only a few tens of gates [7], and is therefore relatively cheap.

achieved. However, the protocols in a content addressable memory are too simple to make efficient use of the communication bandwidth of the accessing mechanism. For only a "few more" gates, one can add a complete microprocessor to each memory cell and have a universal capability there. In this way, the messages can be much higher level than the simple "match and respond" messages of the content-addressable memories.

There has been considerable interest in how to apply these large numbers of processors to the solution of a single task [33]. Since the efficient utilization of a horde of processors will require a lot of communication, sorting networks have been devised [10,87] which allow every processor in an N-processor system to both send and receive a message on every clock pulse. However, it is still not clear how to effectively utilize all of these processors. Later in this thesis, we will make one suggestion ("call-by-future" or "eager evaluation") for keeping all of these CPU's busy.

1.4 Problems with Shared Memory

The hallmark of the Von Neumann computer model is its homogeneous array of read-write memory cells, addressed by a set of contiguous non-negative integers. This memory has been abstracted out from the computer proper as a single separate RAM (random access memory) chip in many current microcomputers. The RAM chip has a set of address lines, a set of data lines, and a read/write line. If the chip is presented with a non-negative integer on its address lines and a "read" signal, the contents of the memory cell addressed by that integer appears on the data lines after a short delay. If the chip is presented with an address and a "write" signal, the data presented to the chip on the data lines is written into the memory cell specified by the address.

A key property of this RAM design is that only *one* address can be presented to the chip at one time, and that address refers to only a single memory cell. This means that if one memory location is being addressed, the others must remain idle. This might not be so bad if only a few memory cells resided on a chip. However, the trend is to put more and more memory on the same chip. A result of this trend is that the fraction of the memory

that is active at any one time is becoming smaller and smaller. This means that in a given number of cycles, less and less of the memory stored in the device can be brought to bear on the problem at hand.

One can counter this argument by saying that the speed of the memory chips has also been increasing, and therefore that this will counteract the previous trend. However, the speed is increasing at a slower rate than the capacity.[3] If we consider the minimum time to examine every location as a figure of merit for a memory module (using parallel access, if the memory allows it), then this figure is increasing with time.

The effect of this trend is to make memory *less* accessible than previously. Of course, it has been argued that few systems take advantage of anything like the bandwidth allowed by the smaller chips, since usually only one of the memory chips is enabled at a time. However, this fact is not something to be proud of.

One can also argue that if the same information must by accessed from many places at the same time, it should be *copied* into separate chips or separate computer systems to avoid the accessing bottleneck. However, multiple copies of memory require multiple amounts of hardware to store. True, the actual cost of the storage itself is very small compared to the accessing network (this is true for the entire spectrum of memory devices from tape drives to memory chips), and therefore copying the whole memory may be no more expensive than copying only the accessing mechanism. Regardless of these costs though, multiple copies of information create great difficulties in keeping those copies consistent, and the communication bandwidth required for this purpose may cost more than keeping only a single, but very accessible, copy.

Multiport memories have been developed which achieve some degree of simultaneous access to more than one memory location in a memory at one time; e.g. there exist small register chips with two completely independent access channels as well as large interleaved memory banks with arbiters, each of which multiplexes access from multiple sources to a

---

3. The speed of a memory chip is roughly inversely proportional to its linear dimension, while its capacity is roughly proportional to its area.

single memory bank. Computer systems using multiport memories and dual processors can achieve a better processor to memory match and more throughput per dollar than a single processor operating on a non-shared memory because the two slow processors are cheaper than the single fast processor. However, the success of cache memories closely tied to CPU's indicates that considerably more can be done in matching CPU performance to memories. This is because accesses from a CPU to a memory are not random and independent, but show a considerable serial correlation. In other words, many of the accesses in a given time period tend to be *close* to one another. By remembering in a fast cache chunks of information which are repeatedly accessed, the communication to the main memory is reduced. This means that for a given memory bandwidth, more CPU's (with caches) can have access to the same memory.

The problem of multiple copies of information raises its ugly head again, though. If multiple CPU's each have a copy of the same information, which is the current one? The answer we propose is to have many CPU's with caches, and to have only one copy of each unit of information and *no* shared memory at all! Thus, if one processor is cacheing a "memory cell", and another wants to access it, it must either ask the first processor to intercede for it, or the first processor must give it up to the second. The first type of interaction is reminiscent of the simple access of a CPU to a memory, while the second is reminiscent of the transferring into fast store a page of memory cells from a backing store in a virtual memory system. The first type of interaction allows a CPU access to any memory in the system, while the second allows the location of information in the system to be optimized, depending upon who accesses it the most.

Thus, the complex address decoding logic of a serial computer which steer pulses from one CPU to one of many memory locations and back can be replaced by a more symmetric arrangement whereby many CPU's send messages among themselves concurrently.

1.5 Real-Time Systems Design

Consider the problem of the designer of a real-time system, a computer system with numerous stimuli which must be responded to within strict time bounds. There are many such systems in existence today, and their number is growing daily. Some examples of real-time systems are an airplane's autopilot, which responds to changes in the plane's course, altitude, or speed; the ignition and fuel injection controllers in some automobiles which respond to changes in throttle position and load; and the distributed computer message switching centers, which must process and re-direct thousands of messages per second.

If computers and all their I/O devices were matched in speed so that a computer could handle exactly one task at a time and *finish it* before starting on the next--all the while meeting the response times required of it--then there would be no problem in allocating either time or space on the system. The currently running task would have the whole machine--all the processor cycles and all the memory locations--until it finished.

In a few fortunate cases, such a system design works well. However, in most cases, this kind of a system leads either to unacceptably long response delays, or unacceptably low utilization of the hardware (i.e. it is too expensive!). Thus, more efficient use of the system resources can be gained through multiplexing processor cycles and sharing memory among the different tasks. The execution of several tasks can then be overlapped with one another.

When many tasks must share the same memory, some management scheme must be instituted in order that this sharing be done harmoniously, and with the least amount of *hogging*. What mechanisms can be used to manage the sharing of memory? If all the different tasks must share the same address space, the simplest method is *fixed allocation*. In this scheme, every task is allocated its storage at system design time, and the task may never use more, regardless of the distribution of stimuli the system is subjected to. This scheme is subject to storage fragmentation because a task always has enough storage for its worst case, whether or not all tasks can achieve their worst cases simultaneously.

A second storage management policy is that of the *pool*, where allocation is dynamic, but its responsibility rests with a central facility and all tasks request storage from it in a uniform way. However, even this policy leads to storage fragmentation. If the pool allows for blocks of any size, it may reach a situation in which a block is needed and there is enough free storage in total to satisfy the request, but that free storage is not available in one contiguous block. Hence, the demand cannot be met. On the other hand, if the pool allows only for blocks of certain sizes, then much storage is lost through rounding requests up to the next block size.

Therefore, for a system to make maximum use of its available storage, it must be able to *re-organize* the storage, i.e. it must rearrange blocks of data in memory so that free space can be made contiguous and hence more available for allocation.

In the simple system which had only one task executing at any one time without pre-emption, each task allocated storage as it saw fit. For example, if the task algorithm were programmed in one of today's higher level languages, it would use a *stack* for local variables and subroutine return linkages, which would grow from one end of the linear array of storage cells.

If the system were extended to use a well-ordered set of interrupt priorities, then it could allow the simultaneous execution of many tasks, all sharing the same stack, so long as the highest priority task finished before the next higher priority one resumed. However, this policy places great restrictions on the freedom of the higher priority tasks to allocate storage, since any object they allocate will be de-allocated before they finish. This means that if they want to return some information--e.g. a buffer of characters read in from some external source--to some lower priority task, it is the lower priority task that must allocate the space for the buffer. Hence storage is again fragmented, since the lower priority task must make a worst case guess as to the size of the buffer needed.

However, a static priority scheduling policy for tasks with hard response time constraints is known not to be optimal in the utilization of processor cycles [63]. A "tightest constraint first" policy can in many cases be very close to optimal [63], but this policy is not a static priority scheme and would upset the delicate coordination of the LIFO storage

allocation with static priority scheduling.

For these and other reasons, such as the desire for coroutines and other control structures more powerful than simple subroutining, one is forced to abandon the single stack method of storage allocation as too restrictive and hard to program. But storage management with more than one stack is a problem. With two stacks, one stack can grow from the bottom while the other grows from the top, but how does one manage three or more stacks?

Some systems get around the problem of the one-dimensional nature of the random access memory through *memory mapping*. This scheme allows every task the illusion that it has the whole memory to itself whereas in reality it has only a whole *address space* to itself. This illusion is implemented by means of a memory map, which is a partial mapping from the address space of each task into the real memory of the computer system. This mapping from an address space to the real memory is not done on a word-by-word basis, because the cost of such a map would exceed that of the complete real memory for the whole address space[4]. Therefore, the mapping is done in larger blocks called *pages*. However, again storage becomes fragmented because whole pages of real memory must be allocated even when only a few words of virtual memory on that page are being used.

The use of a memory map greatly reduces the amount of memory shuffling in multitask systems since information can be contiguous in virtual memory even when it is not contiguous in real memory. However, to reduce such shuffling to a minimum, every task should have its own map. But maps are expensive--both in terms of their hardware cost and in terms of the storage fragmentation they produce.

It is for these reasons that *list memory*[5] is so valuable--not only every task, but *every*

---

4. Such a map could be implemented far more effectively as a content addressable memory, but cheap content addressable memories have yet to appear.

5. We mean by "*list memory*" a memory whose cell adjacency relationships are indicated explicitly with pointers, instead of implicitly through contiguity in the address space. We include small objects having more than two pointers under the definition of list memory, even though the paradigm of list memories, LISP, has only two pointers per object.

list is essentially a little map to the elements in the list, and the position of those elements can be intertwined with elements from other lists or shared with other lists, or even moved, so long as the "map" is updated. Since the list elements are of the same order of magnitude as a word, there is very little storage fragmentation due to "rounding up".

1.6 Why Actors for Real-Time Systems Design?

We have already argued that the standard random access memory is far from optimum as a memory model for a real-time system. We will argue here that the standard *process* model for a task is also inadequate, hence the inadequacy of current hardware scheduling aids such as interrupts.

Most state-of-the-art real-time systems are interrupt driven, with interrupt signals on a vectored interrupt bus causing a context-switch (attention shift) in the CPU. Internally in the software, however, they use a subroutine mechanism to communicate among internal modules which uses a stack as a medium for information exchange and state-saving. On the other hand, actor theory is a theory of message-passing among many modules, and does not distinguish between externally and internally generated messages. It therefore unifies the concepts of interrupt-handling and subroutining. In this theory, an external signal and its corresponding data are packaged together into a *message*, which is then presented to an *actor* for processing. Whether the message is processed immediately or not depends on the scheduling algorithm, and messages generated externally are treated the same as those generated internally by that algorithm. A control stack is not needed because each "return address" is represented explicitly in the naming environment as a *continuation*. A parameter stack is also superfluous because messages are explicitly constructed from heap storage. The only system structure needed explicitly is the *pending event structure*, which the event scheduling algorithm uses to keep track of messages in transit.

Most existing real-time systems use a hardware static priority scheme to filter out high priority from low priority requests for service. This scheme meshes well with the use of a stack for saving the state of interrupted processes, because the priority levels are in a

one-to-one relationship with the levels of saved state on the stack. However, as we have pointed out, this scheme requires that long-term memory for the higher priority tasks be provided for in advance, and this is both wasteful and awkward. It is wasteful because storage must be provided to satisfy the largest request and not the actual request. It is awkward, because the higher priority tasks be programmed in a manner (and perhaps even a language) different from that of the lower priority tasks. There will be little continuity between interrupts to a task because no state for the higher priority tasks may be stored on the stack. Hence, constructs such as "for" loops cannot be used in the programming of these higher level tasks because they store some temporary results on the stack.

A better system would allow every task to be programmed relatively independently of the others, but in the same language. For example, one should not have to know the relative priority of a task at the time it is programmed but one should be allowed to use every construct of the programming language. A system which uses a completely separate address space for each task has most of these properties, but the separate address spaces make it hard for the tasks to communicate with one another. In many such systems, tasks communicate by means of *messages* which are sent and received in much the same way that information is communicated between spatially separated nodes in a distributed system. However, transmitting messages between different tasks in the same computer system boils down to a glorified way of copying the contents of one area of memory to another, and if a large amount of information must be transmitted, the time to send such a message is proportional to the length of the message, including all of its components.

It can be argued that all that copying is not necessary, if one only updates the memory map for the receiving process to reflect the fact that a certain part of memory now contains the message instead of what it used to contain. There are problems with this scheme because mapping is not performed on a per-word basis, and this requires that both the sending and the receiving buffers begin on a page boundary and occupy an integral number of pages. Thus, the average message size will interact with the choice of page size. The result of working out all of these details is most inelegant.

Worse is the fact that the message buffer is now shared between two maps, and if the sender now tries to construct a new message in its message buffer, it will destroy the message being processed by the receiver! So mapping the memory did not achieve what we wanted at all, namely passing a message, but instead achieved the non-goal of passing a buffer of memory cells.

Since we would like to construct and pass messages and not message buffers, we must use a different buffer to construct the messages than the one the receiver is processing, and this requires a great deal of protocol to agree on which buffers are being used and a great deal of synchronization to change buffers. Furthermore, if variable-length messages are being sent and the receiver does not process them in a simple FIFO or LIFO order, the problem of managing the message buffers becomes as big a headache as any in the whole system. The final blow to this scheme is the fact that buffers not used by one pair of tasks in their communication may not be re-used for another communication link.

Therefore, if messages are to be sent and received with a minimum of copying and a maximum of sharing, message buffers must be allocated from a central pool of storage shared by all tasks. (This pool must be shared by *all* tasks because a task may forward a message or point to it as a subpart of another message.) However, once message sharing has gotten this complicated, the responsibility for reclaiming and re-using old message buffers must be taken away from the individual tasks, and given to the central authority, since the individual tasks are not in the best position to know when a buffer is no longer needed. Thus, through a series of logical steps we are now back in the realm of list processing for the management of messages between different tasks in a real-time system.

By proposing a real-time system based on separate tasks sharing a list memory and communicating by passing messages which are stored in the list memory, we solve quite a few of the problems of real-time system design. However, a large problem which results from multiplexing many tasks on a single computer remains. This is the problem of fast context-switching. Modern "mainframe" computers tend to have a large number of registers in the CPU which must be saved on an interrupt and restored upon resumption of that task. In addition, many CPU's have a *cache* memory which is effectively saved and restored

on every interrupt and resume, but perhaps not so obviously. These fast registers and caches speed up the CPU in the execution of one task, but increase the time required to switch tasks. Real-time systems, which must be able to respond quickly to external stimuli, cannot afford to spend a long time saving and restoring the states of tasks, and must therefore minimize the amount of state information needed to represent that task.

Again, computer designers have responded to this problem by implementing *multiple register sets* in the CPU, one per process--in effect implementing a map from task number to register set. In fact, the Texas Instruments 9900 single chip microprocessor takes this scheme to its logical conclusion by keeping all its "registers" in main memory, and switching tasks by changing the CPU register which points to the block of storage allocated for the "registers" of the current task! In this CPU, task switching involves the saving and restoring of only 3 words--the program counter, the task status word, and the register block pointer.

This register mapping scheme certainly solves the problem of long context switching time; a processor using it can switch contexts in only a few memory cycles. However, the whole concept of a task as a process has become more virtual. Memory has been abstracted into an address space and a register has become just another memory cell with a shorter name. One wonders where this process will eventually end, and whether it might be simpler and cleaner to use another conceptual model for programming these systems.

1.7 Continuation-Passing Style

We have been arguing that the standard static-priority, stack-recursive control structures of present day real-time systems are inadequate to deal with truly complex situations involving dynamic priorities and co-routines. The systems that try to handle such situations do so badly because they must allocate multiple stacks with all the problems that they cause.

A heading for all of these issues might be called *focus-of-attention* or the *management* of *attention* because this is what a real-time system must do to respond to its stimuli. Over the past 15 years, researchers in the field of Artificial Intelligence have concerned themselves very much with this issue because for most of the problems in this field, the total amount of computation can be drastically reduced if the attention of the computer is focussed. Since much of this type of computation involves *searching*, the mean search time over many computations can be reduced if the search is performed by looking in the most likely places first, then the next most likely, and so on.

However, the orders of search which are easiest to program--e.g. *depth-first search*--do not usually correspond to the most likely first ordering. Hence the computer must make many shifts of context as drastically different alternatives are examined, one after another. As a result of these needs, A.I. researchers have come to the conclusion that simple recursive, single-stack control structures are not adequate for their requirements. They have found a need for co-routines, generators, and backtracking in order to focus the attention of the computer program upon the currently most promising line to attack its most pressing problem.

Landin [60], Reynolds [73], Hewitt [50], Steele [81,82,84], and others [35,86] have shown that all of these control structures can be modelled very elegantly in a form of programming called "continuation-passing". In the continuation-passing style of programming, the control stack of subroutine return points is not left implicit in the nested structure of the program, but is made explicit by providing an additional parameter in each subroutine argument tuple called the *continuation*. When a procedure A is called from a procedure B with an argument list including C as a continuation argument, procedure A computes its value using the normal arguments passed to it, but instead of "returning" to A, it calls C with the computed value as an argument. But since the body of C encodes all the computations which A would have done on the value returned by B, it is the *continuation* of A after the "return" from B. If one carries this form of programming to the limit, i.e. by everywhere calling a continuation argument instead of returning, then the return points are only pushed onto the stack and never popped. Thus, although the control stack grows to a depth which

is proportional to the length of the computation, it can be eliminated entirely since it is never referenced again.

The control stack is not needed when programming in the continuation-passing style, because it duplicates information already stored in the variable binding environment [50,81,82]. However, this variable binding environment should be a tree-shaped structure to avoid the "FUNARG" problems which would otherwise result. The tree-shaped environments required for the continuation-passing style of programming can be easily implemented in list storage. Since there is no control stack, all storage required by the program can be satisfied by one mechanism--a garbage-collected heap; i.e. one uniform mechanism provides for both the implicit storage required for binding as well as the explicit storage requested by the programmer.

Although a little efficiency is given up by replacing the stack push operation by a continuation creation operation plus a variable binding operation, one immediately gains the flexibility of non-recursive control structures such as co-routines, generators, and backtracking. Furthermore, if one writes continuations in such a way that they accept multiple arguments, then one also gets the effect of returning *multiple values* from a subroutine essentially for free. In this style, a divide subroutine can return both the quotient and remainder from the division process without the usual kludgery involved in handling multiple values.

Continuation-passing style makes the programming of real-time control systems easier, since the logical event causalities of the various tasks are explicit in the text of the program instead of being buried in some scheduler. When a routine is finished performing some computation, it has the flexibility to go directly on to the next computation, whether or not that computation is to be done by the routine which called it. Since the state of the control structure is explicitly represented in the environment instead of implicitly in a control stack, there is very little state in the CPU to change in order to respond to external stimuli quickly. All of the tasks are on the same level, instead of having "interrupt-level" routines, "high-priority" routines, and "background" routines. Finally, since the management of storage for the continuations is handled by the system, the programmer need not worry

about where all of these bits and bytes are being allocated, but only whether the total storage used exceeds the amount available.

1.8 Outline of the Thesis

Chapter 2 presents axioms for actor theory and discusses some theoretical problems posed by them. It also presents a constructive interpreter which is capable of generating all possible computations from an initial configuration of actors. This interpreter is not intended to be used in a real actor system, but only to illustrate more concretely a scheduling mechanism which is consistent with the actor axioms.

Chapters 3 and 4 discuss one of the main problems in implementing an actor system which is subject to real-time constraints--the allocation and recollection of storage. An incremental garbage collection approach is advocated, and a method is exhibited which has the additional property that all allocation, accessing and updating primitives are time-bounded by a constant. Hence, the events in an actor system which uses this technique can also be time-bounded.

Chapter 5 deals with a new problem that comes up in actor systems with large numbers of activities and processors. An activity may be started on the presumption that the result it will eventually return will be useful. However, as other activities progress in parallel with it, this presumption may prove false, and the activity which is now deemed useless must be stopped and its resources returned to the system. One of the best examples of a system which generates activities which may later turn out to be useless is that of an interpreter for an "applicative" (expression-based) language which implements "call-by-future", a parameter binding mechanism which is different from call-by-name, call-by-value, call-by-need, call-by-reference, etc. Call-by-future is implemented by an "eager" interpreter, which spawns a new activity (a "future") for every expression which is an argument to a procedure. Eager evaluation may result in faster response from real-time systems, since an activity does not have to wait until its relevancy is proven before it can be started. The Church-Rosser theorem [21,26], which ensures the invariance of the value of

an expression in these languages regardless of the order of evaluation, can be extended to cover this new evaluation order. Thus, in a language like LISP which has been extended with call-by-future, the value of an expression will be independent of the evaluation order "most" of the time, i.e. whenever the side-effects do not interfere.[6]

In Chapter 5, a garbage collection approach is also advocated for this problem, and a method is found for garbage collecting "irrelevant" (useless) activities incrementally.

---

6. Other researchers [37,28,89] also note that languages without side-effects, e.g. "pure" LISP, are excellently suited for the purpose of representing many algorithms intended for execution on a host of processors since their lack of side-effects eliminates a great source of complexity in parallel execution. However, this kind of parallelism does not implement the most general form of communication between activities. For example, an airline reservation system cannot be implemented in such a language, due to its non-determinate behavior.

2. Laws for Actor Systems

2.1 Introduction

Although there has been much previous work on actor theory [49,85,43,44,51,50,52], the precise semantics of the orderings of events in this theory, the modes of information propagation, and the role of non-determinism have not been clear. As a result, any attempt at a clean realization of the actor concepts in terms of a language was difficult, because these fundamental issues had not yet been resolved. This chapter[1] attempts to clarify actor theory by presenting some axioms that we believe must be satisfied by computations involving communicating parallel processes. These laws restrict the histories of parallel (actor) computations to make them physically realizable. The laws are justified by appeal to physical intuition, and are to be regarded as falsifiable assertions about the kinds of computations that occur in nature rather than as proven theorems in mathematics.

Since the causal relations among the events in a parallel computation do not specify a unique total order on events, actor theory generalizes the notion of a *computation* from that of a *sequence of global states* to that of a *partial order of events*. The interpretation of two unordered events in this partial order is that they proceed concurrently.

Specifications for an actor and correctness assertions for a computation can be given very naturally in terms of events and partial orders of events because partial orders seem better suited to expressing the causality involved in parallel computation than the totally ordered sequences obtained by "considering all shuffles" of the elementary steps of the various parallel processes [74]. Since inference rules can use these partial orders directly, the number of cases in proofs is considerably reduced.[2] We demonstrate some of the utility of these partial orders by using them to express our laws for distributed computations.

---

1. This chapter is an expansion of some of the ideas in the two papers "Laws for Communicating Parallel Processes" and "Actors and Continuous Functionals" by Carl Hewitt and myself.
2. A. Holt [57] and I. Greif [43] were some of the pioneers of event-based reasoning.

We present in this chapter *axioms* for actor systems which restrict and define the causal and incidental relations among events in an actor computation, where an event consists of the receipt of a message by an actor, and results in the sending of other messages to other actors. These axioms do not postulate the existence and fairness of some global scheduler or oracle, even though our constructive model for this theory will use such a global scheduler to ensure that the computations it generates satisfy all of the axioms.

2.2 Event-based vs. State-based Reasoning about Systems

The application of the concept of *state* to sequential systems was a great advance. This concept allowed the future behavior of a system to be completely determined by the abstract state of the system instead of the whole past history of the system. More formally, a state is an *equivalence class* of past histories of a system, all of which are equivalent in the sense that the future behavior of the system given any of these past histories will be identical. In some cases, the infinite (and perhaps uncountable) class of histories can be vastly reduced to a finite set of these equivalence classes, or states. Thus, the state of a system incorporates the "important" part of the past history, where "important" is defined as being relevant to the prediction of future behavior.

Since the concept of global state is such an important and valuable tool to the understanding of systems, why do we give it up? We reject it on both theoretical and practical grounds. Relativity theory tells us that the concept of a global state for a spatially distributed system is ill-defined in the sense that the relative order of many events, and hence the perception of the state, varies with the position (and velocity, etc.) of the observer of the system. Therefore, in order to consistently define a global state, we must specify an observation point and define the time of an event as the instant that the observer observes it. Although this can be done, one would like a more observer-independent description of the behavior of a system. Relativity theory tells us that the direction of causality or the direction of information flow among events is the same for all observers, and hence diagrams of event causalities are theoretically more appropriate for spatially distributed

systems.

Thus, although the concept of state allows us to factor out the irrelevant details of a sequential system's history, partial orders of events allow us to factor out the irrelevant details of the observer of a spatial system's position, velocity, etc. But we would also like to factor out the irrelevant details in the history of a spatially distributed system using a concept similar to that of state. While quite laudable, this goal is hard to achieve. In the case of a sequential system, the concepts of "time" and "behavior" are both well-defined; "time" is a linear order of transitions in the system while "behavior" is a mathematical function of the sequence of all inputs, or equivalently, a function of the current state and the future inputs. But neither concept generalizes for a distributed system.

The concept of a "space-like slice" through the causal connection diagram for the history of a distributed system may be the appropriate generalization of an "instant of time" in a sequential system. These space-like slices are essentially collections of events that are unordered by causality, i.e. they consist of events which could happen simultaneously. Given such a slice, one could identify the local states for each object in the slice. If another causal connection diagram over the same set of system elements were to contain an equivalent slice--namely, one in which the same objects had the same local states--then the histories of both systems (the set of events which preceded the slice) are equivalent, in the sense that the same set of "future" events could be generated. Thus, by defining arbitrary global states (the slices), we can regain the ability to factor out irrelevancies in the past history of a system.

Using this technique, we can compose the histories of two systems using the same configuration of primitive elements.

But we also reject the notion of a history of a system as being a sequence of global states on practical grounds. Suppose that our system consists of n totally independent parts, each having a local state set of size m. Then the global state set consists of m^n different states, a number which for reasonable m and n is totally intractable if each state must be checked for some property. Of course, the parts in any interesting distributed system will

not be completely independent, but even so the size of the total state set will remain an exponential function of the amount of parallelism in the system.

2.3 Events and Actor Computations

In a serial model, computations are linear sequences of global states, and each state in the sequence determines the next state by consulting either a program text (Von Neumann stored program computer), or a finite state control (Turing machine model). In the actor model, we generalize the notion of computation to be a partial order of events in a system, where each event is the transition from one *local* state to another.

The theory presented in this chapter attempts to characterize the behavior of procedural objects called *actors* (active objects) in parallel processing systems. Actors, messages, and events are the fundamental concepts in the theory. Actors interact through one actor sending a *message* to another actor called the *target* (of the message). The *receipt* (and processing) of the message by the target is an *event*, and these receipt events are the basic steps in the actor model of computation.

New actors and messages can be created in an event in the course of a computation.[4] Indeed, almost every message is newly created before being sent to a target actor.

Events mark the steps in actor computations; they are the fundamental interactions of actors. Each event happens instantaneously, i.e. indivisibly, requiring no duration in time. Every event E consists of the receipt of a message, called message(E), by a target actor, called target(E). We will often use the notation

---

3. In Hegel's terms, our thesis is really an antithesis to the thesis of *global state*, especially the proving of properties of parallel systems based on global state transitions. Of course as Hegel pointed out, *synthesis* follows thesis and antithesis, and we have indicated a possible direction for this synthesis in the equivalencing of certain space-like slices. However, since antithesis and not synthesis is our intent, we will argue here for a theory of *events* and *local states* rather than *global states*.

4. The creation of an actor is not itself an event; actors are created as side-effects of other events. We denote the event which results in an actor x being created as the *creation event* for x.

E: [T <~~ M]

to indicate that event E consists of the receipt of message M by target T.

An event is the *receipt* of a message rather than its sending, because the message cannot affect the behavior of its target actor until it is received. If the sender wishes a reply, the message should contain as a component a *continuation*, i.e. an actor to whom any reply should be sent.

Intuitively, the receipt of the message M at the target T makes M's information available to the target for the purpose of causing additional events by sending messages to other actors. The receipt of M by T does not *in itself* cause any change to either M or T; however, T may decide after receiving M to remember all or part of M.

Due to the totality of the "receipt order" for each actor (to be defined later), we may speak consistently about the local state of an actor. This local state is completely encoded as a *vector of acquaintances*, which encodes the names of other actors this actor knows about at this time. A *name* in this vector is just enough information to allow this actor to send a message to the denoted actor.

Therefore, for each event E, we can define $acqs_E(T)$ to be the vector of immediate acquaintances of T "just before" the event E. We now stipulate that this vector is of a fixed, finite length; i.e. that the length of an actor's acquaintance vector is fixed for the life of the actor.

Law of Finite Acquaintances: For all actors x and events E such that x=target(E), the vector $acqs_E(x)$ has finite length. For all events E_1, E_2 such that $target(E_1)=target(E_2)=x$, $length(acqs_{E_1}(x))=length(acqs_{E_2}(x))$.

This restriction is not meant to discourage the use of arrays with flexible bounds. However, they cannot be primitive in our system because in order to satisfy real-time constraints, we want all primitive operations to be (in principle) time-bounded by constants, and all known methods for dealing with such arrays require time growing with the size of the array.

The λ-expressions of Church's λ-calculus [21,26] may be modelled by actors which receive their arguments as messages. In this case, the expressions bound to the *free variables* of the λ-expression x become the acquaintances of the actor modelling x. Due to the properties of the λ-calculus, those acquaintances may not change over time; i.e. if actor y models a λ-expression, then for all events E_1 and E_2 in which y is the target,

$$acqs_{E_1}(y) = acqs_{E_2}(y).$$

In order to implement interprocess communication between parallel processes, it is necessary to use actors whose acquaintance vector changes over time. One purpose of this chapter is to axiomatize the fundamental laws which govern the behavior of such actors.

An important example of an actor whose immediate acquaintances change with time is a *cell*. A cell's acquaintance vector has exactly one element--its *contents*. When the cell is sent a message which consists of the *request* "contents?" and a *continuation* (another actor which will receive those contents), the cell is guaranteed to deliver its contents to that continuation. When the cell receives a message with the command "store y!" and a continuation, the cell forgets its previous acquaintance by updating its acquaintance vector to hold y, and then informs the continuation that the command has been obeyed. The behavior of cells will be discussed later in more detail.

2.4 Partial Orderings on Events

In order to develop a useful model of parallel computation, we have found it desirable to generalize the usual notion of the history of a computation from a sequence of states to a partial order of events. Thus, a history of an actor computation is a partial order which records the causal and incidental relations among events. It is an upper bound on the amount of parallelism that can be used in an implementation, e.g. any two unordered events could be executing concurrently on separate processors. However, there is no requirement that an implementation do this. An actor computation may be simulated by executing the events in any order which is consistent with the partial order defined by the history.

2.4.1 Activation Ordering

One important strict partial ordering on the events in the history of a computation is derived from how events *activate* one another. Suppose an actor x_1 receives a message m_1 in an event E_1 and as a result sends a message m_2 to another actor x_2. Then the event E_2 in which m_2 is received by x_2 is said to be *activated* by E_1, i.e. E_1 is the *activator* of E_2. We call the transitive closure of this "activation" relation the *activation ordering* for a particular actor computation and if E_1 precedes E_2 in this ordering then we write

$$E_1 \leftrightarrow E_2.$$

2.4.1.1 Laws for the Activation Ordering

It is not possible for there to be an infinite number of events in a chain[5] of activations between two given events in the activation ordering of the history of a computation. Stated more formally:

Law of Finite Activation Chains between Events: If C is a chain of events in the activation ordering from E_1 to E_2, then C is finite.

The law of finite activation chains is intended to eliminate "Zeno machines"--machines which compute infinitely fast. For example, consider a PDP10 which executes its first instruction in 1 μsecond, its second in 1/2 μsecond, its third in 1/4 μsecond, and so on. This machine not only could compute everything normally computable in less than 2 μseconds, but could also solve the "halting problem". It could do this by simulating a normal PDP10 running on some input, and if the simulation were still running after 2 μseconds, it could conclude that the simulated machine did not halt on that input.

---

5. A *chain* is a totally ordered subset of a partial order.

It is intuitively reasonable that an actor can construct and send only a finite number of messages in the instant that is an event. Therefore, one event can activate only a finite number of other events. The events directly activated by an event E are called the *immediate successors* of E under the activation ordering, or *immediate activation successors* of E. The set of immediate activation successors of E, written $succ_{++>}(E)$, has the formal definition:

$$succ_{++>}(E) \equiv \{E' |\ E \text{++>} E' \text{ and } \neg\exists\ E'' \text{ such that } E \text{++>} E'' \text{++>} E'\}.$$

Then we have the following law:

Law of Finite Immediate Activation Successors: For all events E, the set $succ_{++>}(E)$ is finite.

We also define immediate predecessors for the activation order in a manner analogous to that used for immediate successors.

$$pred_{++>}(E) \equiv \{E' |\ E' \text{++>} E \text{ and } \neg\exists\ E'' \text{ such that } E' \text{++>} E'' \text{++>} E\}.$$

We now postulate that an event is either an *initial* event, in which case it has no immediate predecessors, or it is activated by a unique predecessor event.

Law of Unique Activators: For all events E, the set $pred_{++>}(E)$ contains either zero or one element.

Each event E has at most one activator event activator(E), because message(E) is the only message received in the event E and because message(E) can only be sent by one event, which is required to be activator(E).

What does this activation ordering look like? Since each event has at most one activator, and no infinite preceding chains, the ordering is a forest of *trees* having the initial events as roots. Since the branching is restricted to be finite at every node, each tree is finitary.

Note that because an event has only one activator, the *join* part of fork-join behavior cannot be analyzed using only the activator ordering. We will see later that having unique activators forces an asymmetry in the analysis of joins because the last event to arrive at the

join is the one which activates the remainder of the computation. Thus, the symmetry of a "joiner" actor[6] is not a foregone conclusion from the basic axioms of actor theory, but must be proven.

2.4.2 Receipt Orderings

Intuitively, the activation ordering can be identified with the notion of "causality", since each event is "caused" by its activator event. However, the activation ordering is not enough to specify the actions of actors with *side-effects*, such as cells. For this reason, we introduce the *receipt ordering* $==>_x$ for an actor x which records the order of receipt of messages sent to x after having been ordered by an *arbiter*. Note that there are only a few primitive actors such as cells and synchronization primitives which actually care about the order in which messages arrive.

2.4.2.1 Laws for Receipt Orderings

The receipt ordering for each actor x is required to be a total ordering on all events which have x as their target. This policy is enforced by *arbitration*, i.e. if two messages arrive in close proximity to x, its arbiter device will arbitrarily decide which is to be received by the actor first.

Law of Total Receipt Orders: If $E_1 \neq E_2$ and $target(E_1) = target(E_2) = x$, then either $E_1 ==>_x E_2$ or $E_2 ==>_x E_1$.

This law states that either $message(E_1)$ is received before $message(E_2)$, or vice-versa.

We note that there is no necessary relation between the order of receipt of two messages at a target and the ordering of their activators. Suppose that events E_1 and E_2 both have the same target x. In a serial computation, $E_1 ==>_x E_2$ would imply that $E_1 <+>E_2$, but in a parallel computation, E_1 and E_2 could be parts of two separate processes

---

6. Later, we introduce a particular kind of "joiner" actor called a "gluer".

unrelated via ++>. Furthermore, the fact that activator(E_1) precedes activator(E_2) in the computation is no guarantee that $E_1 ==>_x E_2$ because message(E_1) could take a longer route than the message(E_2), or be delayed by an arbiter.

If an actor x is created in the course of a computation, then prior to any given message which it receives, it could only have received finitely many other messages.

Law of Finitely Many Predecessors in the Receipt Ordering: If an actor x is created in the course of a computation, and target(E)=x, then $\{E' | E' ==>_x E\}$ is a finite set.

The above law is used to guarantee that the process of repeatedly taking the precursor of an event will eventually stop, i.e. no receipt ordering is an infinite descending chain.

Given an event E_1: [T <~~ M_1] and an event E_2: [T <~~ M_2], there are only a finite number of events between the two in the receipt ordering $==>_T$. Stated more formally:

Corollary: For all events E_1, E_2 such that target(E_1)=target(E_2)=x, $\{E | E_1 ==>_x E ==>_x E_2\}$ is finite.

This law eliminates anomalous behavior like the following: a cell receives an infinite sequence of "store!" commands: "store 1!", "store 1/2!", store 1/4!", "store 1/8!", etc. and then receives a "contents?" request. What is it to reply to the continuation? Zero? But zero was never explicitly stored into the cell!

The Law of Finite Chains in the Receipt Ordering allows us to define *immediate* predecessors and *immediate* successors for this ordering in a manner similar to the one used for the activation ordering. Since the Receipt Order Law guarantees that the receipt order for each actor is total on its domain, immediate successors and predecessors are unique, when they exist. If an event E has an immediate predecessor in $==>_{target(E)}$, it will be called the *precursor* of E and will be denoted precursor(E).

One of the simplest examples of an actor which depends upon its receipt ordering for well-defined behavior is the *cell*. The cell is the actor theory analogue of the program variable in modern high-level programming languages in that it has a value which can be changed through assignment. This value is encoded as the cell's single, changeable

acquaintance which is initialized to the name of some actor when the cell is created. A cell responds to two types of messages, "contents?" requests and "store!" commands. When a cell receives a request [contents? reply-to: c], the cell sends the name of its acquaintance to the actor c. When a cell receives a command [store! y reply-to: c], it forgets its previous acquaintance, memorizes y as its new acquaintance, and then sends an acknowledge message to c.

We will discuss cells more formally in a later section.

2.4.3 The Combined Ordering

Since the events in any legal actor computation must be consistent with both the activation and receipt orderings, they must be consistent with the transitive closure of the union of the two. Hence, we introduce the concept of the *precedes* relation, "-->", which combines the restrictions of both of these relations.

Definition: "-->" is a binary relation on events which is the transitive closure of the union of the activation ordering "++>" and the receipt orderings " $==>_x$ ", for every actor x. In mathematical notation,

$$--> \equiv (++> \cup \bigcup_x ==>_x)^+.$$

In order for "-->" to function as a precedence relation, the next law requires that the activation and arrival orderings be consistent. The Law of Strict Causality states that there are no cycles allowed in causal chains; i.e. no event in any history of any actor system precedes itself. Stated more formally,

Law of Strict Causality: For all events E, it is not the case that E-->E.

This law does not follow from the properties of the activation and receipt orderings, and counterexamples can be easily generated.

Now the immediate predecessors and successors of an event in the combined ordering are the unions of its immediate predecessors and successors in the constituent orderings. Therefore, an event has at most two immediate predecessors--its activator and its precursor--and at most a finite number of immediate successors.

We would like to formalize the intuition that between any two events which are causally related, there are only a finite number of events in the causal chain which links the two. We therefore have the following law:

Law of Finite Intermediate Chains in the Combined Order (Discreteness of the Combined Order): Given two events E_1 and E_2 in an actor computation, there does not exist an infinite chain in --> between E_1 and E_2.

This law has a corollary which is even stronger:

Corollary: Given two events E_1 and E_2 in an actor computation, there do not exist an infinite number of events between then in "-->"; in other words, the set

$$\{E \mid E_1 \dashrightarrow E \dashrightarrow E_2\}$$

is finite, for every choice of E_1 and E_2.

Proof: For any arbitrary choice of E_1 and E_2, let S denote the set described in the statement of the corollary. Suppose that S were infinite. Now S has a spanning tree in "-->" with E_1 as its root, so S contains an infinite tree. What is the maximum number of branches protruding from any arbitrary node in this tree? The immediate successors in "-->" of a node are the immediate successors of that node in "++>", plus the successor of that node in the receipt ordering for that node's target, if such a successor exists. It then follows from the Law of Finite Immediate Activation Successors that the immediate successors of a node in "-->" must be finite, hence the number of branches in our tree protruding from any node must also be finite. Hence the tree is finitary. But then by König's Lemma, this infinite tree must contain an infinite chain. Since this contradicts our Law of Finite Intermediate Chains in the Combined Order, the corollary stands.

QED

While this law and corollary would seem to be a consequence of the discreteness laws for each of the constituent orderings, plus the consistency requirement for the combined ordering, it is in fact independent of those laws, as the counterexample in a later section will show.

2.4.3.1 Fork-Join Behavior

In programming languages for parallel processing, it is necessary to provide primitives by which a process can "fork" by splitting into several processes which can later "join" together again. This allows for the processing of one branch of the fork to overlap with the processing of the other fork, thus allowing for a reduction in the time to complete the overall task, assuming that sufficient hardware is available for such concurrent processing.

The parallel (collateral) evaluation of the arguments for a procedure call provides a very common and natural example of such fork-join behavior. Suppose, for example, that we are interested in computing the value of "a^2+b^2" for some a and some *b*. In order to reduce the computation time, we would like to evaluate a^2 and b^2 in parallel before summing the results. To evaluate these two arguments to "+" in parallel, the evaluation process must split into two sub-processes, each of which evaluates one argument. When both have been computed, they must be brought back together to form an argument pair which is then sent to the "+" procedure. This process of combining the results of the two parallel processes is a form of *synchronization* between the two processes, because more than likely one will finish its evaluation before the other and therefore have to wait.

We can simulate this form of synchronization with a primitive actor called a *gluer*, which accepts messages from two different sources, glues them together into a single message, and then sends them to a continuation which was supplied when the gluer was created. A more formal description of a gluer is given below.

Although a gluer requires an arbiter in front of it to keep from receiving two messages at the same time, and hence getting confused, its behavior is symmetrical. The particular order of receipt of those messages does not matter since the gluer does not activate any other event until it has received messages from both of its senders, i.e. the last message received activates the sending of the combined message to the continuation, regardless of the source of that last message.

Gluers allow us to factor the work of an actor which receives parameters from several different sources into two parts: a gluer which receives the different parameters and binds then together into a single message, and the computational part of the actor which performs the intended operation on the multiple operands which the gluer has brought together. In an actor simulation of the *data-flow* computational model [28], every multiple-input operator would require a gluer to glue one *token* from each input arc into one composite token which would trigger the actual computation.

However, a gluer is different from a two-input dataflow operator because it has only one input port through which it can process messages, and these messages are arbitrated to arrive in a total order. Therefore, although the gluer is entirely symmetrical in that its output is independent of the order of receipt of the two different flavors of messages, it is inherently a serial device, like every other actor, which is capable of receiving only one message at a time. Because of its ability to glue together different messages which arrive at different times, i.e. it gathers together data presented to it serially, the gluer is a sort of "serial-to-parallel" converter.[7]

We now analyze an example of fork-join behavior using this *glueing* primitive.

2.4.3.2 Formal Description of a Gluer

There is a primitive actor, called *create-gluer*, such that whenever it receives a message of the form [sink:S reply-to:R], it creates a new gluer actor G, whose *sink* is S, and sends it to R. G then accepts messages of two forms: [left: x] and [right: y], where x and y are arbitrary actors. If G receives a message of the form [left: x] and has previously received a

message of the form [right: y], it sends a message of the form [reply:[x y]] to S. If G receives a message of the form [right: x] and has previously received a message of the form [left: y], it sends a message of the form [reply:[x y]] to S. Thus, a message of the form [left: x] is a "left-hand component" and a message of the form [right: y] is a "right-hand component" of a final message to the sink S. Note that if in a computation, m left-hand messages and n right hand messages are sent to the same gluer, then the gluer sends m*n messages to its sink, these m*n messages consisting of all the combinations of left hand and right hand messages.

Figure 1 below shows an event diagram of the general kind of gluer described above, while Figure 2 shows the diagram for the collateral evaluation of the expression "a^2+b^2". We note that in the latter case we have two possibilities for the event diagrams, depending upon which multiplication sub-expression returns a value to the gluer first.

---

7. Because of this restriction on actors that they can receive messages only one at a time, one might conclude that they are not as powerful or as fast as a data-flow operator, which can accept data on all its input ports "simultaneously". The truth is, in order to physically perform the synchronization required, whether in actor theory or dataflow, the control information about which operands are ready and which are not must all propagate to a single point in space at which, according to the assumptions of actor theory, the signals will all arrive in some order and not simultaneously. Normally, an arbiter that decides which signal arrives first takes time inversely proportional to the time difference of the arrivals. However, since the result of a gluer is the same in either case, it should not need an arbiter on its input; i.e. since a gluer does not reveal its decision about the order of arrival, it might be able to use a different circuit than a standard arbiter. This circuit might even be faster because the theoretical arguments against fast arbiters would not apply to gluers. This argument is a gross simplification of some of the ideas of quantum theory, but it should retain some validity.

Fig. 1. Event Diagram of a Gluer

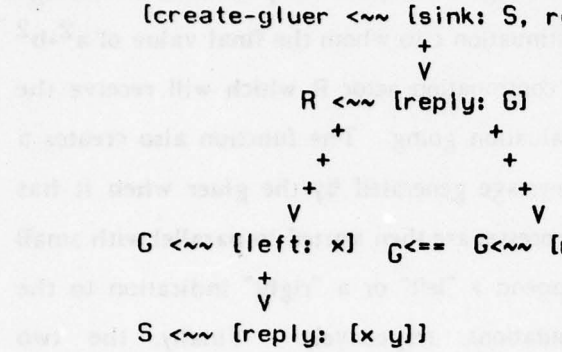


Fig. 2. Parallel Evaluation of an Expression

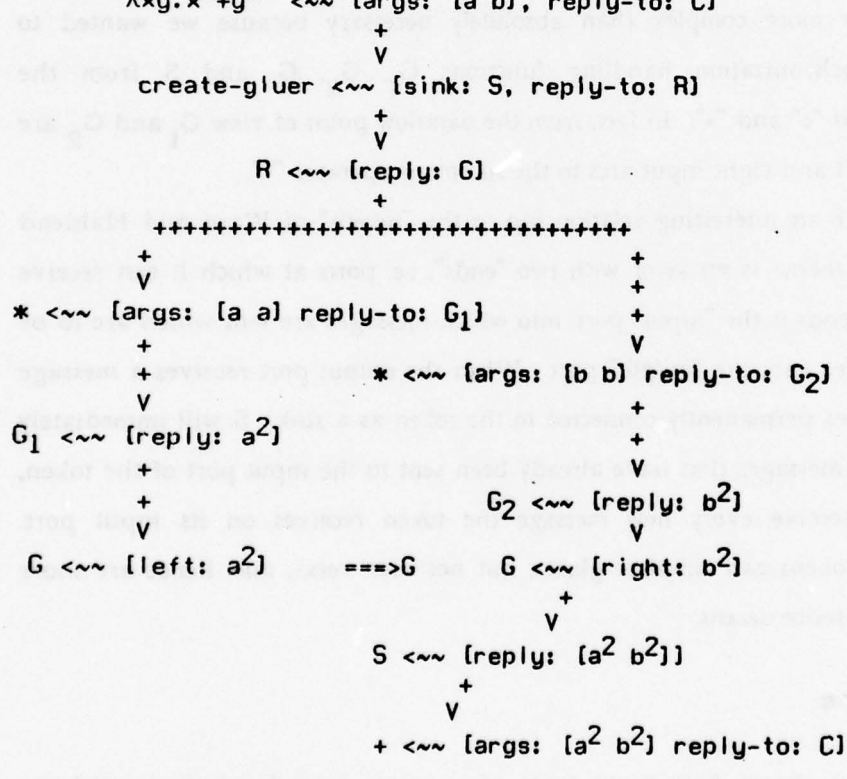


Figure 2 requires some explanation. The original function x^2+y^2 is sent a message consisting of the arguments a and b and a continuation c to whom the final value of a^2+b^2 should be sent. The function then creates a continuation actor R which will receive the newly created gluer and start the parallel evaluation going. The function also creates a continuation actor S which will handle the message generated by the gluer when it has glued the two sub-results together. The sub-processes are then started in parallel with small subsidiary continuations G_1 and G_2 which append a "left" or a "right" indication to the results of the first or second sub-computations, respectively. Finally, the two sub-computations both reply to G through G_1 and G_2 and the glued result is passed onto the "+" actor by means of the continuation S.

This example is more complex than absolutely necessary because we wanted to separate out the synchronization handling functions G_1, G_2, G, and S from the computational functions "*" and "+". In fact, from the dataflow point of view G_1 and G_2 are acting simply as the left and right input arcs to the summing operator "+".

These gluers bear an interesting relationship to the "tokens" of Ward and Halstead [96,45]. One of their *tokens* is an actor with two "ends", i.e. ports at which it can receive messages. One of the ends is the "input" port, into which messages are sent which are to be retrieved from the other end, the "output" port. When the output port receives a message [output-to: S], S becomes permanently connected to the token as a sink. S will immediately receive the backlog of messages that have already been sent to the input port of the token, and will henceforth receive every new message the token receives on its input port. Halstead claims that tokens can simulate gluers, but not vice versa, and hence are more primitive. See [45] for more details.

2.4.4 Activities

Hewitt [49,50] has shown how many types of program control structures such as procedure invocation, recursion, backtracking, and parallel evaluation of arguments can be easily analyzed as patterns of message-passing among the actor-like modules of a

programming system. We would like to characterize one of the most common of these patterns, the request-reply pattern, as a *goal-directed activity*.

Intuitively, a goal-directed activity starts with a *request* event, in which an actor receives a message containing 1) a request for a computation, 2) some arguments for that computation, and 3) the name of an actor--the *continuation*--which is to receive the reply when it is ready. The activity then consists of all events which result from the request, directly or indirectly, up to and including a *reply* event. The *reply* event consists of the receipt of a reply message by the continuation actor specified in the request event for the activity.

More formally, let E--≥ denote the set of events which follow E (including E itself) and --≥E denote the set of events which precede E (including E) in the computation.

$$E\text{--}\geq \equiv \{ E' \mid E \dashrightarrow E' \text{ or } E = E' \}$$

$$\text{--}\geq E \equiv \{ E' \mid E' \dashrightarrow E \text{ or } E' = E \}$$

Then the goal-directed activity A_Q corresponding to a request event E_Q in a computation is the set of events which follow E_Q but precede any reply E_R to the request; i.e.

$$A_Q = E_Q\text{--}\geq \cap \bigcup \{\text{--}\geq E_R \mid E_R \text{ is reply to } E_Q\}$$

Goal-directed activities embody the notion of the nesting of activities that is produced by the standard subroutine-calling of conventional programming languages. For example, a request to the "tangent" procedure might result in requests to the "sine" and "cosine" procedures, and replies from them, before the tangent of the argument is returned as the reply to the outer request.

Several things should be noted from this definition. First, there may be no reply whatsoever to a request, which means that the goal-directed activity consists of a single event, the request. Since a goal-directed activity is meant to include only those events which eventually led to the reply, there may be none if no reply was ever made. This type of behavior is to be expected from functions which are partial, due to oversight or

incompleteness.

However, just because the goal-directed activity is empty does not mean that no events are occurring. Many events may be taking place which contribute to no request's reply and hence are wasted. These lines of computation can by definition be eliminated without affecting the results of goal-directed activities. The problem of detecting and eliminating this wasted computation is considered in a later chapter of this thesis.

It should also be noted from the definition that some goal-directed activities consist of exactly two events, the request and the reply, with no intervening events. This means that no requests to sub-activities needed to be sent in order to process the request; the answer was available immediately. We call these activities *primitive activities*, because they cannot be further decomposed; the buck stops here. Primitive activities are necessary, because they are where the real computational work is done.

Finally, the definition for goal-directed activities allows the possibility that several replies may be made to the continuation of a request. This is because in some patterns of passing messages, an activity might act like a non-deterministic generator, returning every answer which was plausible, rather than a single correct one. However, this may not be an interesting pattern if the number of replies is unlimited, because since no acknowledgments are required from the receiving actor to continue the replies, the pattern allows for no way of stopping the replies.

2.4.4.1 Concurrent Goal-directed Activities

Intuitively, several activities may be proceeding in a computation at the same time. We can formalize this through the notion of *concurrent activities*. Two activities are concurrent if their request events are unordered, i.e. if their request events are concurrent. An interesting situation arises if concurrent activities overlap, i.e. share some events. This can happen if (and only if) the activities both involve sending messages to the same shared actor. If two concurrent activities involve only *pure* actors, and these pure actors are freely copied to avoid arbitration bottlenecks, then goal-directed activities are *properly nested*,

meaning that two activities are either disjoint, or one is a subset of the other.

2.4.4.2 Homomorphisms of Computations

The notion of activities allows one to vary the level of detail used in modelling a real system with actors. Whereas in a crude model an activity might be *primitive*, with no intermediate events between a request and the corresponding reply, a more detailed model could use an activity with a whole host of intermediate events and sub-activities. If the internal workings of this activity were independent from the rest of the computation, then suppressing this extra detail should not detract from an understanding of the rest of the system.

2.4.5 Actor Creation and the Laws of Locality

In many models for distributed computation the ensemble of processes or actors is fixed at the time the computation is initiated. The communication patterns within this fixed collection of objects can be ascertained (or at least bounded) before the computation starts, and therefore every object knows at the time the computation is started exactly which other objects it may send messages to and which other objects it may receive messages from. As a result of this restriction, no actor *names* need ever be passed in messages. If an actor A ever needs to distinguish the messages it sends to an actor B from all the other actors which might also send messages to B, A need only include a small integer which would distinguish it from the other actors who might also send messages to B. Then B can use this small integer to look up in a small, constant, local table generated at initialization time to determine who sent the message. Thus, *global* actor names would not be needed at all.

However, in the general actor theory presented here, new actors may be created in the course of a computation. This ability, while adding considerably to the power of actor systems, also adds new dimensions to their subtlety.

The creation of new actors at run-time implies that the names for some actors are not known at initialization time. Hence, if these new actors are ever to be sent messages by any actor other than the one which created them, it must be possible to pass their names around in messages.

By far the greatest use for these newly created actors is that of *continuations*. To implement the standard call-return sequence in an actor system the caller of a "subroutine" will include an additional continuation parameter in the message it sends to the subroutine. This *continuation* is an actor which will receive the value computed and returned by the subroutine; hence, it plays the role of the "return address" in less sophisticated systems. Since in most cases, the behavior required of the continuation for a particular call is not known until just before the call, the continuation must be newly created when the call is made (i.e. when the parameter-continuation message is sent).

This ability to create new actors in the midst of an actor computation and pass their names around means that not only may new nodes be added to the network connecting the actors, but the topology of the network connecting the existing actors may change over time as actors are introduced to each other and forget old acquaintances.[8] But even worse, it makes no sense to ask of such a network what the global connection pattern looks like *even in theory*. This is because the connection pattern changes over time and because there is relativistic ambiguity about the precise ordering of changes not already ordered by the general precedes relation. One would have to define the relativistic notion of a "space-like slice" through the computation and speak of the connection pattern relative to one of these slices in order to gain a consistent meaning to the topology of an actor computation at a given "point in time".

Definition: The target(E) and the message(E) and their immediate acquaintances will be called the immediate *participants* of the event E. The immediate participants of an event

---

8. This does not contradict the fact that the *length* of an actor's acquaintance vector does not change over its lifetime. It only means that one acquaintance may be forgotten in the process of acquiring a new one.

are exactly those actors which can be "known" in the event without the sending of any more messages.

participants(E) ≡ {target(E),message(E)} ∪ $acqs_E$(target(E)) ∪ $acqs_E$(message(E))

We then have the intuitive corollary of the law of Finite Acquaintances that only finitely many objects participate in a single event.

Corollary: For each event E, participants(E) is finite.

Intuitively, the creation of an actor must precede any use of it. In order to state this intuition as a law, we must be more precise about when actors are created. For each actor x which is created in the course of a computation, we shall require that there is a unique event creation(x), in which x was created.

Let created(E) be the set (possibly empty) of actors created by the event E, i.e. the set of actors which claim E as their creation event. Note that x cannot participate in creation(x) because x does not come into existence until after creation(x) has occurred.

Definition: created(E) ≡ {x|creation(x)=E}.

Law of Creation before Use: If an actor x is created in the course of a computation and E is an event with target x, then creation(x) --≥ activator(E).

The intuition that a single event create only finitely many objects is formalized as follows:

Law of Finite Creation: For each event E, created(E) is a finite set.

2.4.6 Laws of Locality

Our intuition tells us that causality in the physical world is local, that there is no "action at a distance". The actor model conforms to this intuition in the sense that all causality is mediated through messages. In other words, information in an actor computation is transmitted by, and only by, messages.

The most fundamental form of knowledge which is conveyed by a message in an actor computation is knowledge about the existence of another actor. This is because an actor A must "know about" another actor B, i.e. know B's *name*, in order to send B a message. However, an actor can know an actor's name only if it was either created with that knowledge or acquired it as a result of receiving a message. In addition, an actor may send a message to another actor conveying only names the first actor already knows; i.e. it may not make up a name out of thin air and send it in a message as a genuine name.

The rest of this section formalizes these intuitions as laws which legal actor computations must obey. In an earlier section, we introduced the notion of an actor's *acquaintances* and stipulated that at no time could an actor remember the names of more than a finite number of other actors, i.e. its acquaintance vector was finite. We now want to be more precise about how an actor's vector of acquaintances may evolve over the course of its local time.

An actor is given a finite initial vector of acquaintances when it is created.[9] We require that every element of this initial vector be a participant of the actor's creation event, since intuitively an actor can initially know about only its parents, acquaintances of its parents, and its siblings. Therefore, we have the following law:

Law of Initial Acquaintances: If an actor z is the target of an event E and E is the first event in the receipt ordering for z, then

$$acqs_E(z) \subseteq participants(creation(z)) \cup created(creation(z)).$$

The acquaintance vector of an actor may change as a result of the messages it receives. When it receives a message, it may add to (or replace one of the elements of) its acquaintance vector any actor's name mentioned in the message. It is also allowed to forget acquaintances at any time. An actor can also remain *pure* by refusing to change its

9. Some actors are primordial; i.e. they exist at the beginning of the computation. If for uniformity's sake they need a creation event, the initial event which started the computation will serve.

acquaintance vector. Most actors remember very little of what they have been told. For example, a *cell* has exactly one acquaintance, its *contents*, which it can be asked to divulge or replace on command.

The following law encodes the intuition that the most an actor may learn from an event are the names mentioned in the message and the new actors created in the event.

Law of Precursor Acquaintances: If an actor z is the target of an event E and E has a precursor in the arrival ordering of z, then

$$acqs_E(z) \subseteq participants(precursor(E)) \cup created(precursor(E)).$$

As we have noted above, an actor is restricted in what other actors it can send messages to. In particular, an event E may activate an event E' only if the target of E' is a participant of E or created in E and each actor mentioned in the message of E' must also be a participant of E or created in E.[10] This gives rise to the following law:

Law of Activator Acquaintances: For each non-initial event E,

$$target(E) \in participants(activator(E)) \cup created(activator(E))$$

and

$$message(E) \in participants(activator(E)) \cup created(activator(E)).$$

These *locality* laws rule out "broadcasting" protocols in which messages are sent to every actor in the system.[11] This is because the phrase "every actor" is not well-defined in a model which allows the creation of new actors, but has no global states in order to pin down precisely which actors are in existence at any given "time". Broadcasting protocols are not inconsistent with the other axioms of actor theory, but making their semantics precise would

---

10. Recall that the participants of an event include the acquaintances of the target and the message.

11. However, a message distribution center can be built so that a single message can be sent to every actor registered with the center.

add a source of indeterminacy in addition to that introduced by the arbitration which makes the receipt ordering total for every actor.

2.4.7 Actor Induction

Using the different ordering relations on an actor computation--the activation ordering, the receipt orderings for every actor, and the combined precedes ordering--one can prove properties about the computation through *actor induction*. Actor induction, a form of structural induction on the structure of the actor computation, consists of two parts. Suppose that one is trying to prove property P of every event in an actor computation. One must first prove that P is true of the initial event E_0. Then if one can prove that P is true of E, assuming that P is true of every immediate predecessor of an arbitrary event E in the given ordering, then we may conclude that P is true of every event in the actor computation.

For example, suppose that one wanted to prove an invariance property P about a certain *actor* A in an actor computation. One need only prove that P is true of A "immediately after"[12] A's creation event, creation(A), and that if for every event E in which A receives a message, P is true of A immediately after precursor(E) implies that P is true immediately after E, then P is true of A immediately after every event in which A receives a message. Since events in which A receives a message are the only ones which can affect A, P is true of A for the whole computation.

This example makes use of an important special case of the following principle:

Law of Precursor Order Induction: If property P is true of the initial event E_0 in an actor computation, and if for all $E \neq E_0$, P(precursor(E)) implies P(E), then P is true of every event in the computation.

---

12. If a property is true "immediately after" an event E, then it is true for every immediate successor of E in the combined ordering.

Recall that the *precursor* of an event E is the previous event in which target(E) received a message, or the creation event for target(E), if E is the first event in which target(E) receives a message. Hence the receipt ordering for every actor is a sub-ordering of the precursor ordering. Thus, in our example using the receipt ordering for A above, we let P be trivially true for all events in which A is not the target, and the other events (with the exception of creation(A)) form precisely A's receipt ordering.

Precursor order induction is useful for doing "data type inductions" to prove that certain properties of data objects are preserved. Properties of control structures and properties of computations which do not involve side-effects are proven using *activation order induction.*

Law of Activation Order Induction: If property P is true of the initial event E_0 in an actor computation, and if for all $E \neq E_0$, P(activator(E)) implies P(E), then P is true of every event in the computation.

For example, every property of a *serial* computation--one in which the precedes ordering is linear--can be proven using only activation order induction.

Complex properties or properties like synchronization which involve both the activation and receipt orderings require full actor induction over the combined precedes ordering.

Law of Combined Order Induction: If property P is true of the initial event E_0 in an actor computation, and if for all $E \neq E_0$, P(activator(E)) *and* P(precursor(E)) together imply P(E), then P is true of every event in the computation.

2.4.8 Cells

The behavior of cells can be axiomatized by positing a primitive actor *create-cell*, which generates new cells upon request. These generated cells are new in the sense that they are not *shared* with any previously generated cell, i.e. a change to the newly generated cell will have no effect on previously generated cells and vice versa.

Creation: An event of the form:

E_1: [create-cell <~~ [initial-contents: i, reply-to: c]]

activates exactly one event, which has the form:

E_2: [c <~~ [reply: n]],

where n is the newly created cell. Furthermore, created(E_1)={n}, and creation(n)=E_1, which says that n is the only actor newly created in E_1. Thus, each cell returned by create-cell differs from all previously created cells because those cells have different creation events.

Use: Cells recognize only messages of two types:

[contents? reply-to: c] and [store! y, reply-to: c].

Intuitively, a cell has exactly one acquaintance, its *contents*, which may be queried or updated by *contents?* and *store!* messages. We will use the notation $contents_E(n)$ to denote the acquaintance of the cell n for the event E in which n receives a message.

The behavior of a cell can be completely characterized in terms of this *contents* function, as follows.

$contents_E(n)$ ≡
    if E is the first event in the receipt ordering for n
        then i, where
            [create-cell <~~ [initial-contents: i, reply-to: c]]
            is the creation event for n
    else if precursor(E): [n <~~ [store! x, reply-to: c]]
            then x
    else $contents_{precursor(E)}(n)$.

Contents: An event of the form:

E_1: [n <~~ [contents? reply-to: c]]

activates exactly one event, which has the form:

E_2: [c <~~ [reply: $contents_{E_1}(n)$]],

and created(E_1)=created(E_2)=ø.

Update: An event of the form:

E_1: [n <~~ [store! y, reply-to: c]]

activates exactly one event, which has the form:

E_2: [c <~~ [reply: done.]],

and created(E_1)=created(E_2)=$\varnothing$.

2.4.9 Busy Waiting and Fairness

Busy waiting is a synchronization method used in some multiprocessing systems where either the only communication between processors takes place through shared memory, or a processor cannot depend on the others to "wake it up" when the others are ready to signal it.

Consider the example in Figure 1 below in which a processor A must wait for a processor B to reach a certain point before processor A can proceed. A shared memory cell S is initialized to a value known to both processors. Then processor A goes into a tight loop, continually checking the contents of S for a change. When processor B is ready to signal A, it stores a new value into the shared cell S. Processor A will notice that the value of S has changed and will proceed out of its loop.

Busy waiting requires that the memory shared between the two processors be arbitrated so that the one processor does not try to read the contents of the cell during the same cycle in which the other is changing those contents. (Otherwise, the read might produce garbage.) The axioms of actor theory imply the existence of such an arbiter. However, an arbiter can be *unfair* in the sense that it always gives priority to one processor or the other, and in the worst case, may *lock out*, or *starve*, one processor completely. Much effort has gone into the problem of specifications for the fairness of the arbiter which schedules the requests processed by the memory, and elaborate algorithms for fair synchronization have been developed.

Fig. 3. Busy Waiting on a Cell

```
cell S init(0),          % S is initialized to zero.  %

%                Code for Processor A.              %

loop:    if contents(S)=0
         then goto loop
         else ...proceed...

-------------------------------------------------------------------
%                Code for Processor B.              %

 ... Calculates something which A needs ...

S := 1;          % Tell A that we're done.  %
                 % Assume B is the only processor writing into cell S. %
```

The actor model requires no such notion of scheduling or fairness to prove that lockout or starvation is impossible, at least at the level of elementary message receipts. Why? By definition, a completed actor computation has no undelivered (i.e. unreceived) messages outstanding. Thus, every message "eventually"[13] gets through ("neither rain nor sleet..."). That a message gets through within a finite number of steps follows from the "no infinite descending chains" property of the receipt order for every actor. Therefore, between any two messages which are received by a cell, at most a finite number of others can be received. In our example above, between a "contents?" message from processor A and the "store! 1" message from processor B, only a finite number of other messages will be received, and hence the cell's contents will eventually change. Furthermore, between the receipt of the "store! 1" message from B and the next "contents?" message from A, only a finite number of messages can be received and hence A will eventually detect the change in the cell's contents. However, the "length of time" (i.e. the number of receipts processed by the cell) required to synchronize using this simple method is not bounded by any computable function (using only these basic axioms of actor theory). So, although busy waiting is guaranteed to work, it may not be a satisfactory synchronization method.

---

13. Perhaps only after an unbounded amount of time.

We have just shown how the underlying message transmission mechanism of actor theory satisfies the weakest reasonable form of fairness: every message sent is eventually received by the target after it has received at most a finite (but a priori unbounded) number of other messages. However, this weak fairness of the actor transmission may not be shared by higher level protocols built using this simple mechanism. Thus, for more complex objects such as monitors [54] or serializers [51], fairness properties must still be proven.

2.4.10 Discreteness -- A Counterexample

One question that comes up in relation to the Actor theory axioms we have presented is whether or not they are independent, i.e. whether any axiom can be proved using the other axioms. In particular, the question arises as to whether the discreteness of the precedes relation is a consequence of the discreteness of the activation and precursor orderings.

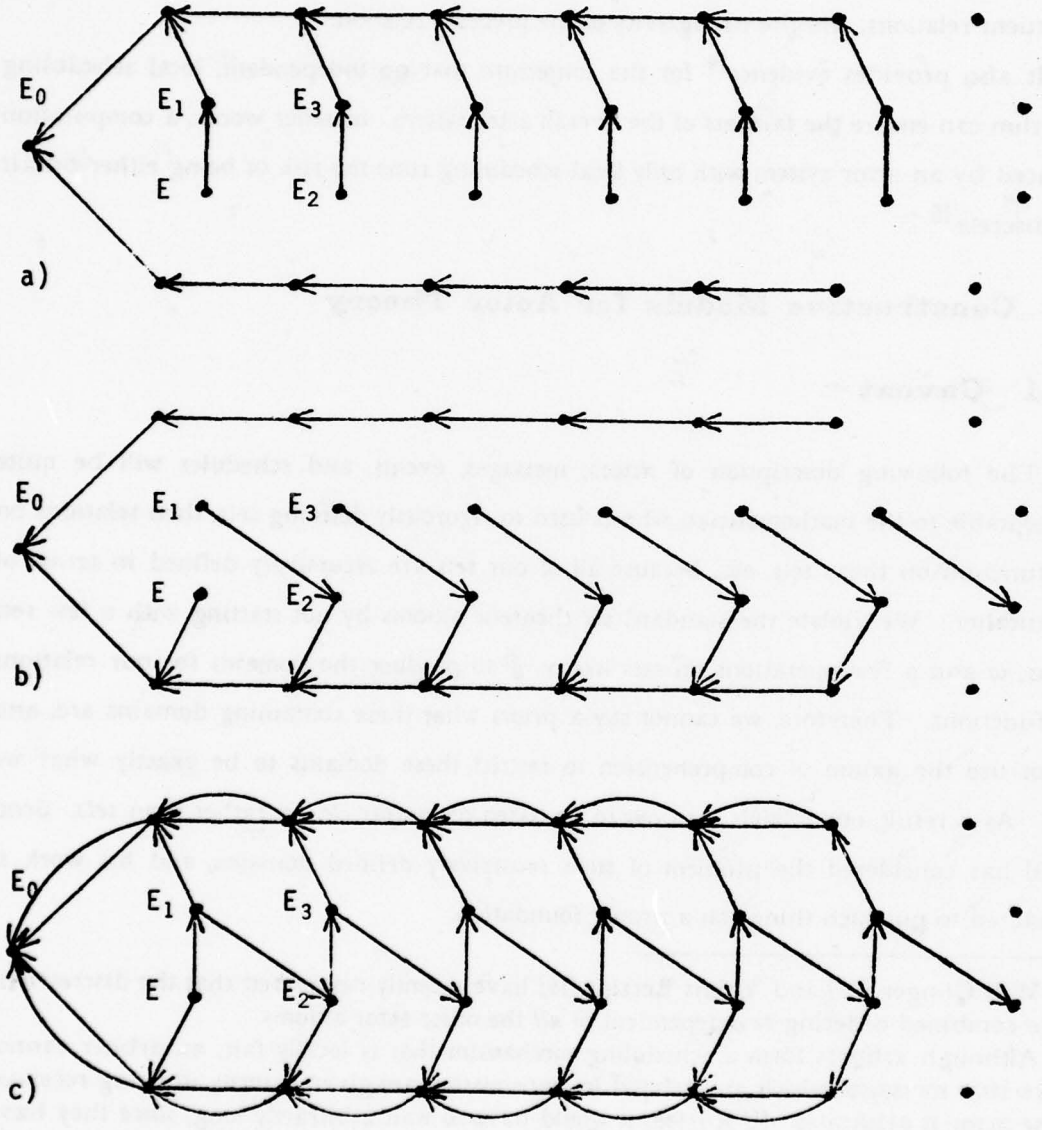
The answer to this question is *no*, because there exist two finitary directed rooted trees over the same infinite set of nodes, such that the closure of their union is a strict partial order, yet the partial order is not discrete.

A diagram for this counter example appears in Figure 4. Figure a) shows the first finitary tree over the nodes, figure b) shows the second finitary tree, while figure c) shows their union. (Only a skeletal set of arcs actually appears; the rest are implied by transitivity.) The root node for both trees is called E_0, and each tree spans all the nodes. Notice that there are no cycles in c), yet there are an infinite number of nodes in the partial order between E_0 and E. Hence c) is not discrete, even though both a) and b) are.

If we were to interpret c) as an actor computation, we could choose a) as the activation ordering and b) as the precursor ordering. However if we examine carefully the structure of c), we notice that there is something strange going on. E's activator event is E_1 which must be preceded by E_2 in the precursor ordering. Now E_2 cannot be the creation event for target(E_1) since the creation event for the target of an event must precede or be the activator for the event. Therefore E_0 must be the creation event for the target of E_1.

Likewise, creation(target(E_3)) must also be E_0. Continuing in this manner, we see that the creation events for all the E_i's must be E_0. But this contradicts the axiom that E_0 can create only a finite number of different actors. Therefore, the locality and finite creation restrictions (to be defined below) rule out this diagram as a legitimate actor computation. (Notice that c) is almost symmetrical, so that interchanging the interpretation of a) and b) does not help.)

Fig. 4. Counter-example to the Discreteness of the Combined Order



This counterexample shows that the discreteness of the combined order does not necessarily follow from the discreteness of the activation and receipt orderings. The significance of this is that if the discreteness of the precedes relation follows from the other axioms, it must depend on more than the discreteness, rootedness, and finitariness of the two constituent relations, and the irreflexivity of the precedes relation.

It also provides evidence[14] for the conjecture that no independent, local scheduling algorithm can ensure the fairness of the overall actor system. In other words, a computation produced by an actor system with only local scheduling runs the risk of being either unfair or indiscrete.[15]

2.5 Constructive Models for Actor Theory

2.5.1 Caveat

The following description of actors, messages, events, and schedules will be quite unacceptable to the mathematician who is used to rigorously defining sets, then relations on sets, function on those sets, etc., because all of our sets are recursively defined in terms of one another. We violate the standard set theoretic axioms by not starting with a few sets like $\varnothing$, ω and a few operations on sets like $\times$, $\mathcal{P}$ to produce the domains for our relations and functions. Therefore, we cannot say a priori what these containing domains are, and cannot use the axiom of comprehension to restrict these domains to be exactly what we want. As a result, our models turn out to be based on *proper classes* rather than *sets*. Scott [78,79] has considered the problem of such recursively defined domains, and his work is considered to put such things on a proper foundation.

---

14. Will Clinger [23] and Valdis Berzins [14] have recently discovered that the discreteness of the combined ordering is independent of *all* the other actor axioms.

15. Although arbiters form a scheduling mechanism that is locally fair, an arbiter cannot ensure that messages which are delayed in transmission are given priority in being received by the actor it arbitrates. (If it tried, it would have to wait arbitrarily long, since they have no idea what messages are in transit to its actor.) Therefore, this mechanism cannot guarantee that every message will eventually be received.

Actor theory as a first order set theory is guaranteed to have a model if it is consistent.[16] However, the model guaranteed by this theorem is not very useful for understanding actors because it is produced from the purely syntactic material of the defining axioms. We would like to produce more constructive, intuitive models which give more insight into the nature of actors, as well as proving that this theory is consistent.

Unfortunately, due to the extreme generality of the theory, with its mutually recursively defined sets, we are pushed to the limit in our ability to put the constructive models themselves on a sound mathematical basis. However, we do have another recourse--a computational model using recursively defined data-types such as LISP's S-expressions. Even though we may be hard pressed to give a proper mathematical interpretation to such objects, they certainly exist and we may compute with them. Thus, if a computational model for actors can be produced, it will prove the consistency of actor theory, assuming that LISP (or whatever such language) is consistent.

2.5.2 Constructive Models

We conceive of two computational models of actor theory, one taking cells as primitive concepts, the other using only constructions which do not involve side-effects. While the cell model is simpler and quite intuitive for anyone who has programmed a computer, it does nothing to explain what a cell is, since it takes the cell as primitive.

We will first present the cell model, and then the pure model.

2.5.3 The Cell Model for Actors

An *actor* in the cell model consists of a triple <*name,script,acquaintances*>, where *name* is an identifier which uniquely determines the actor, *script* is a constant program text in some language, and *acquaintances* is a constant vector of *storage cells*, each of which holds a

16. This does not imply its completeness, as there may be several models which disagree with each other on unimportant details.

pathname (roughly a pointer) to another actor.

Names for actors serve to distinguish each actor in a computation from every other actor. A convenient way to accomplish this is if the name of an actor is a pair <*creation,index*>, where *creation* is the creation event for the actor, or the distinguished indicator NIL if it is an initial actor, and *index* is a finite non-negative integer which distinguishes this actor from its *siblings* (other actors claiming the same creation event). In addition to distinguishing, actor names also *identify*, in the sense that an actor's name determines the actor, hence its script and its vector of acquaintance cells. (However, the actual acquaintances themselves can only be determined relative to a given event in the actor's receipt ordering, since they can change from one event to another.)

Scripts for actors are finite programs in some programming language which are executed upon the receipt of a message by the actor. Upon invocation, the script may create a finite number of new actors and messages and send them off to other actors. It may also modify some of the cells in the local acquaintance vector to forget their current contents or remember some new contents. It may reference various components of the message. However, it may not loop and it may not parameterize a reference to the message or one of its acquaintances; i.e. it may refer to acquaintance 3 but not acquaintance i. Therefore, since the script is constant and finite, it can refer to only a bounded number of storage cells in the acquaintance vector and hence our restriction on acquaintance vectors to have fixed, finite lengths is no hardship.

The acquaintance vector plays a role in actor theory similar to that of the local binding frame in current higher level language semantics. The cells of an actor's acquaintance vector are initialized to hold the initial acquaintances of the actor when the actor is created. These cells may be updated as a side-effect of an event having the actor as its target, but are completely private to the actor and inaccessible to scrutiny or change by any other actor. In other words, the acquaintance cells are not actors themselves; they have no names and can receive no messages. When they are updated as a side-effect of an event, their updating is indivisibly tied up with the event; before the next receipt of a message by the actor, the new acquaintances are well ensconced in these storage cells.

An *initial configuration* for an actor computation is a finite set of initial actors, i.e. actors which are primordial since they lack a proper creation event, and a single pending event in which an initial message is sent to one of the initial actors. An *actor computation* $C=<\mathcal{E},"-->",E_0>$ derived from an initial configuration is a set of events $\mathcal{E}$, strictly partially ordered by the relation "-->", with a distinguished element E_0, which is the least element of $\mathcal{E}$ with respect to the ordering "-->". Each non-least element E is a quadruple <T,M,A,P>, where T is the *target* actor which receives the *message* M in the event E, where A is the *activator* event which sent M to T, and where P is the *precursor* event, i.e. the previous event in T's receipt ordering (or T's creation event, if E is the first event in which T receives a message). Finally, E_0 is the event in which the initial message of the initial configuration is received by its intended target actor; i.e. $E_0=<T_{initial},M_{initial},NIL,NIL>$.

The participants of the initial event E_0 include the *actor* $target(E_0)=T_{initial}$, which has no creation event because there are no events before E_0, yet this actor must exist before it receives a message. However, there may be other initial actors (the other participants of E_0), and E_0 can be conveniently assigned as their creation event without contradicting our axioms. This convention has the advantage that no additional law is required to specify that the number of initial actors is finite, since we already have a law requiring that only a finite number of actors may be created in one event. If more than one initial actor were allowed, a separate axiom to this effect would be required.

Although we have described actor computations as static, already completed objects, they can be analyzed as having been built recursively, starting with E_0 from the simple initial configuration. E_0 creates some new actors, sends messages and activates new events, which in turn send messages and activate other events, and so on. A complete actor computation is the *limit* of this process; it is the final structure which is achieved after all events have occurred and all messages have been received.

This is entirely analogous to the construction of the natural numbers from the empty set. In this construction, we have an initial configuration--the empty set--and a process for taking one configuration to a new one--adding the successor of an element already obtained--and define the natural numbers as the limit of this process.

However, unlike the situation with the natural numbers, wherein the process for converting a configuration into its successor was uniquely determined, the process for converting an actor configuration into its successor is not single valued, and the various possibilities may even be inconsistent (unable to coexist in the same computation). This means that there need be no single, unique actor computation derived from an initial configuration. This non-determinism is due entirely to the arbitration required to determine a receipt ordering for all actors. For example, if two unordered messages arrive at an actor, the order in which they are processed is not determined, yet this order can drastically affect the outcome. For example, if the messages were requests to an airline system for a reservation on the last seat on a flight, the order of receipt would determine who was assigned the seat and hence who would be affected if the plane crashed. Therefore, we must either talk about the *set* of possible computations derived from an initial configuration, or else talk of the computation as proceeding non-deterministically. We will initially take the second approach.

2.5.3.1 Partial Computations

In order to see that an actor computation is isomorphic to the limit of a process which starts from an initial configuration and continually adds new events, we must consider what the intermediate states, which we call *partial computations*, look like.

In a partial computation, there are some messages which have been sent but not yet received, i.e. some events have been activated, but have not yet occurred. These messages in transit, these pending events, must be explicitly represented in the partial computations. There are several alternatives available in choosing a representation for these pending events, such as sets, queues, etc., but we will ignore this problem for a moment.

A partial computation is a triple $<\mathcal{E},-->,P>$, where $\mathcal{E}$ is the set of events which have already occurred, "-->" is the precedence relation built up so far among those events, and P is the "pending event" structure which represents the activated events that have not yet occurred. The initial configuration is then $<\{\},\{\},P_0>$, where P_0 represents the single

pending event wherein the initial actor is sent an initial message.

The process which takes a partial computation to a larger computation we call the *interpreter*. Intuitively, the interpreter removes a pending event from the pending event structure and causes it to occur, i.e. it adds to to the event set and adds the appropriate edges to the precedes relation. In so doing, it adds to the pending event structure all the new events that the occurring event activates. If the pending event structure becomes empty, i.e. if there are no pending events, then the computation is complete and the interpreter reaches a fixed point.

In many cases, however, the computation will be infinite and the pending event structure will never become empty in any finite amount of time. We would like to consider all (finite or infinite) fixed points of the interpreter for an actor system *accessible from* the initial configuration of the system to be the actor computations which are derived from that initial configuration. Since in general the individual steps of the interpreter are non-deterministic, there will be many of these different fixed points.

Serious questions arise about the fairness with which the interpreter selects events from the pending event structure. If the interpreter picks an element from the pending set randomly and independently at every stage, then the probability that a pending event will never occur approaches zero. In other words, in the space of all possible interpreter choice sequences, the set of unfair sequences has measure zero. However, the set of unfair sequences is not necessarily empty! Therefore, this "random" interpreter cannot be a model for actor computations because it satisfies our actor axioms only *probabilistically*; i.e. it admits of unsatisfactory computations, although they have only measure zero in the whole set of generated computations.

Suppose now that we choose a strict first-in, first-out (FIFO) queue for our pending event structure. Then an event, once activated, will never have to wait more than a finite number of steps to occur, since the length of the queue is always finite, and the pending event cannot lose its place in the queue (i.e. be pre-empted). This model satisfies the axioms of actor theory, in particular the discreteness axiom for the precedes relation, and therefore is a logical model of the theory.

Ward and Halstead [96] propose the FIFO model for the pending event structure of a restricted actor theory in which the precursor ordering is always implied by the activation ordering. This restricted actor theory requires no arbiters since there is no freedom in the order of receipt of messages.[17] Since the FIFO model is non-pre-emptive, an event, once scheduled, will occur within a finite number of interpreter steps. Thus, the limit of this process will produce the (essentially unique) completed actor computation which follows from the given initial configuration. Figure 5 shows a FIFO event scheduling algorithm.

However, a strict FIFO queue rules out other modes of behavior, other scheduling strategies, which are also acceptable models of actor theory. For example, using the FIFO model makes the interpreter and hence every computation strictly deterministic, since there

---

Fig. 5. FIFO Actor Interpreter

```
t := 0;              % Keeps track of last scheduled event.  %
S(0) := E0;          % Initial event is only one initially scheduled. %
for i=0 to ∞         % The clock ticks forever.  %
do begin
   let T=target(S(i)), M=message(S(i)), A=activator(S(i));
   % Find precursor for this event by scanning back.  %
   for j=i-1 by -1 until target(C(j))=T or C(j)=creation(T)
   do nothing;
   let P=C(j);  % This is the precursor event for the current event.  %
   let E=<T,M,A,P>;
   % Update Partial Order with this new event.  %
   PO := PO ∪ {A-->E, P-->E};
   % Compute new events to schedule.  %
   let eventlist=match(M,T,P);
   % Schedule these new events.  %
   for e∈eventlist
   do begin
      t := t+1;          % Compute next open slot.  %
      % Schedule it there with E as the activator.  %
      S(t) := <target(e),message(e),E>;
      end
   C(i) := E            % Event E is complete.  %
   end;
```

---

17. They make the additional assumption that if an event dispatches two messages, they are appended to the FIFO queue in the order given by the script of the event's target.

is never any ambiguity about the order in which pending events are processed. Since actor theory requires only that all messages arrive in a finite amount of time, but prescribes no other conditions on the order of arrival of those messages (except when the receipt of one message precedes the sending of another), there may be computations derived from an initial configuration which are not isomorphic to that generated by the FIFO model, yet these computations still satisfy the axioms of the theory. Therefore, we would like a model for actor theory which is more general, i.e. which produces more computations than the FIFO model, without producing any unfair computations.

The scheduling model for actor theory presented below has the appropriate characteristics.

Our scheduling model[18] represents the pending event structure by an *instantaneous schedule*, which posts the scheduled time of execution for every pending event. At the time an event is activated, a time slot is non-deterministically chosen so as not to conflict with any previously scheduled events. This non-deterministic strategy purposely leaves gaps in which events may be scheduled which are activated later. It also retains the property that once a pending event has been scheduled, it may not be pre-empted or re-scheduled. Therefore, at the time it is activated, a pending event is given a bound on the amount of time it must wait before it is executed. Hence, a pending event is guaranteed not to wait forever for execution, and thus this scheduling strategy is free of individual starvation (*fair*).

An *instantaneous schedule* consists of a non-negative integer i, and a pair of partial functions C_i, S_i, whose domains are subsets of the non-negative integers. The integer i denotes the current event number, a crude clock which indicates how many cycles the interpreter has been through since it started with the initial configuration. The first partial function C_i has the set of events $\mathcal{E}$ as its range, and for every interger $0 \le j < i$, $C_i(j)$ is the event which occurred at time j, if any. The second partial function S_i has the set of

---

18. Some of the ideas for this scheduling model were formed during conversations with Eliot Moss.

pending events as its range, namely triples of the form <T,M,A>, where T is the target of the message M which was sent as a result of the activating event A, and $S_i(j)$ denotes the pending event which is scheduled for time j, if any.

Now since the intended interpretation of the instantaneous schedule $<i,C_i,S_i>$ is that the events in the range of C_i have already occurred, while those in the range of S_i are only scheduled, we need a formal *consistency requirement* on instantaneous schedules which ensures that this is the only interpretation. This consistency requirement states that an event occurs at time j if and only if it was scheduled to occur at time j. More precisely, for all j such that $0 \leq j < i$, either $S_i(j)$ and $C_i(j)$ are both undefined, or they are both defined and they refer to the same event, i.e. target $(S_i(j))$=target$(C_i(j))$, message$(S_i(j))$=message$(C_i(j))$, and activator$(S_i(j))$=activator$(C_i(j))$.

Our interpreter I takes as input an instantaneous schedule whose clock reads time i and non-deterministically produces an instantaneous schedule for time i+1. Thus, the computations which can be derived from an initial schedule S_0 can be characterized by the various limits $I^n(S_0)$ as n approaches infinity.

An interpreter step consists of one of the following two cases. Let $<i,C_i,S_i>$ be the input instantaneous schedule. If $S_i(i)$ is undefined, then no event is scheduled for time t=i, so return the instantaneous schedule $<i+1,C_i,S_i>$. In other words, the interpreter idles on this step.

If $S_i(i)$=<T,M,A>, then for time t=i an event is scheduled in which the actor T receives the message M which was sent in the event A, i.e. A activated this event. To complete the current event, we need its *precursor*. The precursor can be found by searching the C-vector from t=i-1 backwards to creation(T) until either an event P is found such that target(P)=T, or creation(T) is reached, in which case let P=creation(T). In either case, let the new event E be <T,M,A,P>.

Now the script for the actor T will tell how message M is to be interpreted using the current acquaintance vector of T, i.e. the script will indicate what new actors to create, what new events to activate, and how to update T's acquaintance vector. The script creates these

new actors, updates T's acquaintances, and produces a finite list L of n pairs $\langle T_i, M_i \rangle$ which specify the events which E should activate. The interpreter must now *schedule* these pending events by choosing a sequence $\langle t_1, t_2, \ldots, t_n \rangle$ of distinct non-negative integers such that $t_j > i$ and $S_i(t_j)$ is undefined, for all j, $1 \leq j \leq n$, where n is the length of the list L. The number t_j indicates the time at which the pending event L_j should occur, which may not be earlier than the current event, and which may not conflict with a previously scheduled event.

Once these events have been scheduled, this step of the interpreter is done, and it returns the instantaneous schedule

$$\langle i+1,\ C_i \cup \langle i, E \rangle,\ S_i \cup \bigcup_{j=1}^{n} \{\langle t_j, \langle T_j, M_j, E \rangle\rangle\} \rangle$$

as its result. The interpretation of this interpreter step is that event E has occurred at time i, and activated the n events which are scheduled at times t_j, with targets T_j, messages M_j, and activator E.

Figure 6 exhibits such a scheduling model for actor computations which uses arrays of cells for acquaintance vectors.

Our scheduling model is not the most efficient possible for generating the legal actor computations from an initial configuration. In particular, the pending event schedule could probably be more efficiently implemented with a *priority queue* [1,92], which would allow the interpreter to skip over empty slots when nothing is scheduled. However, our model is simple and precise, and so it serves our purpose here.

2.5.3.2 An Example of Constructive Interpretation

We would like to illustrate the operation of the interpreter with a trivial example. Consider an actor system with only two actors, A and B. In the initial event for the computation of this system, actor A sends actor B two different messages, M and M'. Because of the totality of the receipt ordering for actor B, the messages must arrive either in the order M, M' or in the order M', M.

Fig. 6. The Cell Model for Actor Computations

```
for i=0 to ∞    % The clock ticks forever.  %
do if S(i) is defined   % Is there an event scheduled for time i?  %
   then begin
        let T=target(S(i)), M=message(S(i)), A=activator(S(i));
        % Find precursor for this event by asking the target.  %
        let P=if most_recent_target_event(T) is defined
                then most_recent_target_event(T)
                else creation(T);
        % Create the event node.  %
        let E=<T,M,A,P>;
        % Update Partial Order with this new event.  %
        PO := PO ∪ {A-->E, P-->E};
        % Apply script of target to message to produce new events and
          update acquaintances of target.  %
        let eventlist=apply(script(T),M,acquaintances(T));
        % Schedule activated events.  %
        for e ∈ eventlist
        do begin
           let j=i+guess();     % Guess a time in the future.  %
           while S(j) defined
                do j:=j+1;      % find first free slot thereafter.  %
           % Schedule e with E as activator.  %
           S(j):=<target(e),message(e),E>
           end
        C(i) := E;
        end;
        .
```

A trace of the scheduling model on this computation is given in Figures 7-10. The interpreter starts the whole computation with only one event scheduled, the event in which A receives a message M_0 to initiate the rest of the computation. To execute this event, the interpreter scans backward through the previously completed events (of which there are none) to find the most recent event in which A received a message. There is none, since this is the first event, so this event will have no precursor event. The first event E_0 is then created having A as the target, M_0 as a message, NIL as the activator and NIL as the precursor. This event is then entered into the partial order with no relationships to any other events because there are no other events yet. The interpreter then matches M_0 to A's script to determine what new actors to create and what new messages to send in order to activate more events. Since A is to send two messages to B upon receipt of M_0, the interpreter schedules a time for the occurrence of these two future events, where B receives

M and B receives M'. Suppose for example that the pair <B,M> is scheduled first for time t=6. This means that there are still empty slots in the schedule for times t=1,2,3,4,5. When the interpreter schedules the pair <B,M'>, it can choose one of these empty slots or a slot after t=6, but it cannot choose the slot at time t=6 because <B,M> is already scheduled then. Suppose that the interpreter chooses the slot t=5 for the pair <B,M'>. Both new events are scheduled by registering them in the "S" vector. Finally, the event E_0 is registered in the "C" vector, indicating that its execution is complete and the first cycle of the interpreter is done.

The next four cycles of the interpreter (with t=1,2,3,4) do nothing because no events are scheduled at those times. On the fifth cycle, the pair <B,M'> is scheduled to occur and the interpreter looks back through the "C" vector for events with B as a target. It finds none, and since B was not created in the course of a computation, there is no precursor for this event, either. The event E_1 is created having B as its target, M' as its message, E_0 as its activator, and NIL as its precursor. This event E_1 is entered into the partial order with the single relationship E_0-->E_1 because E_0 activated E_1. Then the interpreter matches B's script against the message M' to decide what new events E_1 should activate, and these events are scheduled. E_1 is registered as complete, and the fifth interpreter cycle is done.

On interpreter cycle t=6, B is scheduled to receive M. The interpreter scans backward through the completed event list "C" looking for events having B as a target. The first such event it finds is E_1, which it just completed. E_1 becomes the precursor event for the new event E_2, which has B as its target, M as its message, and E_0 as its activator. The partial order is updated to contain the new event E_2 and the new relationship E_0-->E_2 (because E_0 activated E_2), and the relationship E_1-->E_2 (because E_1 is the precursor of E_2). Any events activated by E_2 are then scheduled, and the computation proceeds from there.

Fig. 7. Constructive Example: t=0

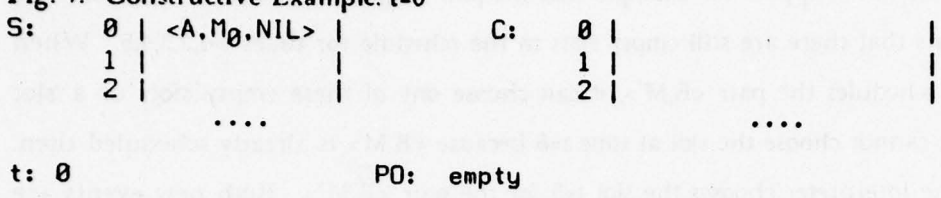


Fig. 8. Constructive Example: t=1

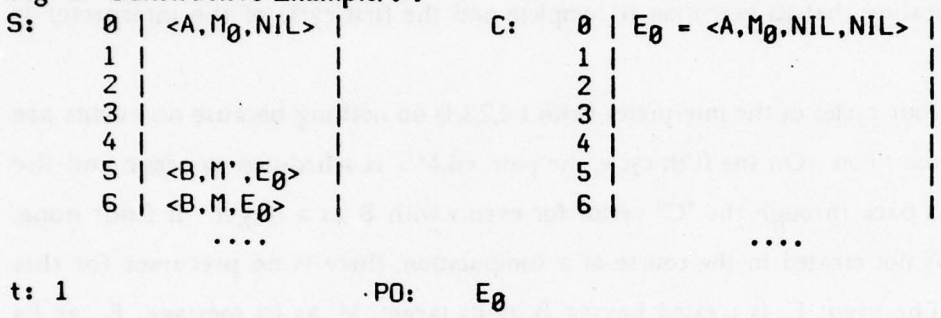


Fig. 9. Constructive Example: t=6

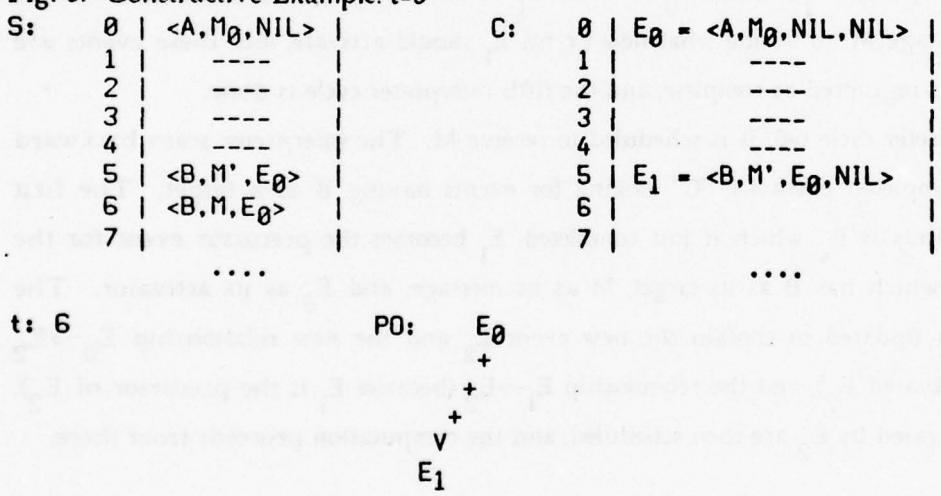
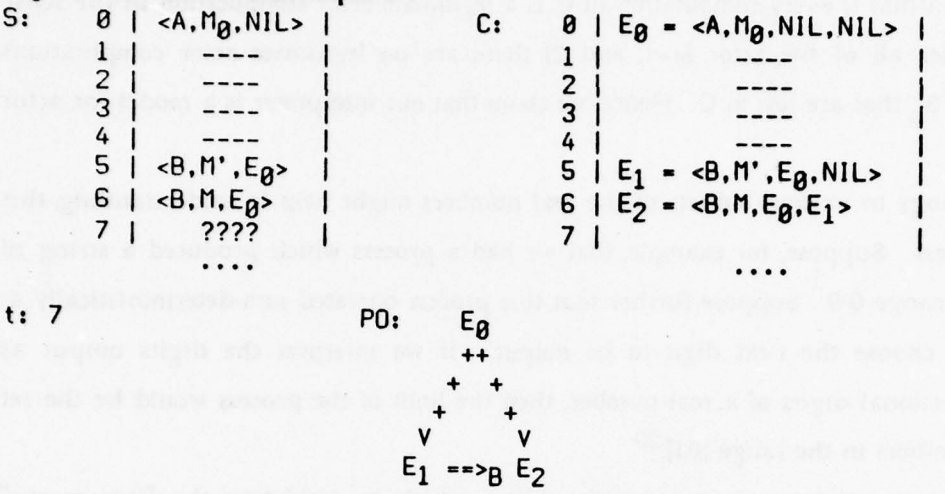


Fig. 10. Constructive Example: t=7



2.5.4 Sets of Actor Computations

We initially made the assumption that our interpreter I nondeterministically produced a new instantaneous schedule from an old one. One can define a corresponding interpreter I' which operates on *sets* of instantaneous schedules.[19] For every instantaneous schedule S in the input set, I' produces all possible schedules I(S) in the output set. Furthermore, for every instantaneous schedule S' in the output set, there exists a corresponding input schedule S, such that S' is one of the schedules derived in one step from S by I. Thus, I' is a single-valued function on the power set of finite instantaneous schedules.

The complete set of actor computations derived from the initial schedule S_0 may be described as the limit of $I'^n(\{S_0\})$ as n approaches infinity, i.e.

$$C = \lim_{n\to\infty} I'^n(\{S_0\}).$$

Thus, C is a set of instantaneous schedules which have become infinite in all possible ways.

19. G. Plotkin [70] has investigated *powerdomains*, similar to power sets, which can be used to make our recursively defined sets of schedules well-defined.

We claim that 1) every computation in C is a legitimate actor computation in the sense that it satisfies all of the actor laws; and 2) there are no legitimate actor computations derived from S_0 that are not in C. Hence, we claim that our interpreter is a model for actor theory.

An analogy to various subsets of the real numbers might help in understanding this limiting process. Suppose, for example, that we had a process which produced a string of digits in the range 0-9. Suppose further that this process operated non-deterministically at each step to choose the next digit to be output. If we interpret the digits output as successive fractional digits of a real number, then the limit of the process would be the set of all real numbers in the range [0,1].[20]

Suppose now that we have an actor system which is simulating the "fair merge" operator of dataflow systems. This operator accepts inputs from two different sources, and produces an output stream consisting of the merged sequence of inputs. However, if this merge operator is to be "fair", it may not decide after a certain time to *ignore* all inputs from one of its sources and take inputs only from the other. If we code the decisions of the merge operator as a finite string of 0's and 1's, where a 0 means that the corresponding output came from the left input source and a 1 means that the corresponding output came from the right input source, then the fairness criterion means that the decision string may never terminate with an infinite string of 0's or 1's.

The set of computations derived from such an actor simulation of a fair merge operator will be in a 1-1 correspondence with the set of infinite strings of 0's and 1's. Again interpreting these strings as infinite fractions between 0.0 and 1.0, but this time coded in binary, we have a correspondence between the set of computations and the set of non-terminating binary fractions. Since the terminating fractions are only of measure zero in the set of all real numbers, *most* arbitrary merge sequences are fair. However, the set of actor computations of this simulation is carefully constructed to avoid the non-fair

---

20. This example requires only *finite* branching at each point, whereas our constructive interpreter effectively branches countably infinitely at every step.

sequences.

Since the arbiter on the front of every actor is essentially a fair merge operator which merges the unordered messages from a wide variety of sources into a single totally ordered sequence, the set of computations for almost every actor system must be constructed with same subtlety as the set for the fair merge operator in order that they satisfy the discreteness requirements of the precedes ordering.

2.5.4.1 Reduced Sets of Actor Computations

Once the set of all actor computations which can be derived from an initial configuration has been constructed, the information about the pending event structures and the instantaneous schedules can be thrown away. The pending event structure is not needed because in the limit, there are no pending events. The instantaneous schedule is also no longer needed because all it does is encode an existence proof that the precedes order is capable of a monotonic embedding into the non-negative integers; the particular embedding does not matter. Thus, the set of all actor computations is partitioned into equivalence classes of instantaneous computations that share the same partial orders. Hence, this partitioned set is isomorphic to the set of completed computations (partial orders) which follow from the initial configuration.

2.5.5 The Pure Model for Actors

We would now like to give a "pure" model for actors in which the acquaintances of an actor do not have to be kept in storage cells which are updated as the computation progresses. We do this to avoid the circularity of explicating cells in terms of acquaintance vectors of cells. We eliminate these cells (at some cost in "efficiency") by re-computing on each interpreter step what the current contents of the target's acquaintance vector should be. This is done through a procedure which recurses backwards along the target's precursor chain and when it reaches the target's creation event, it gets the target's initial acquaintance vector. The procedure then unwinds by going forward along the precursor chain,

re-executing enough of the target's script at every event in order to compute the new acquaintance vector for the next event. Upon completion of this process, the target's current acquaintance vector is available so that the target's script can receive the current message.

We illustrate this process by showing how it works in the case of a simple storage cell. Recall that a storage cell has exactly one acquaintance--its contents. It is created with some initial contents, and it responds to two types of messages--"contents?" and "store!". Conceptually, in the cell model for actors, when a storage cell receives a "contents?" message, it simply looks in its acquaintance vector and delivers up what it finds there to the continuation which was supplied. Again, in the cell model, when the cell receives a "store!" message, it smashes the current contents with the new value which was supplied.

Figure 11 gives a script for such a cell which uses an array of cells as an acquaintance vector.

Figure 12 shows a *pure* (side-effect free) model for a cell. It uses a subsidiary function "lookup" which is not part of the cell's script, but is a meta-function used by the interpreter. (This is because a script cannot refer to *events* in the computation, only actors.)

Fig. 11. A Cell Model for a Cell

```
cell-1: (=> [message: M]
            (cases M
                   (=> [contents? reply-to: C]
                       activate <C, [acquaintance(0)]>)
                   (=> [store! x reply-to: C]
                       acquaintance(0) := x;
                       activate <C, [done!]>)))
```

Fig. 12. A Pure Model for a Cell

```
%       Initial contents of cell-2 is NIL.       %
cell-2: (=> [message: M]
            (cases M
                   (=> [store! x reply-to: C]
                       activate <C, [done!]>)
                   (=> [contents? reply-to: C]
                       activate <C,contents(P)>)))
%       P is precursor of this event.    %
contents(P) = if P=creation(cell-2)
                 then NIL
                 else if message(P)='[store: x reply-to: C]
                         then x
                         else lookup(precursor(P))
```

3. Storage Management and Garbage Collection

EXODUS 12

22 And ye shall take a bunch of hyssop, and dip *it* in the blood that *is* in the bason, and strike the lintel and the two side posts with the blood that *is* in the bason; and none of you shall go out at the door of his house until the morning.

23 For the LORD will pass through to smite the Egyptians; and when he seeth the blood upon the lintel, and on the two side posts, the LORD will pass over the door, and will not suffer the destroyer to come in unto your houses to smite *you*.

King James Version of the Bible

In this chapter, we consider a problem which arises in the implementation of actor systems intended for real-time applications. This problem is the management of acquaintance vectors, messages, and the like. Although many ad hoc schemes could be contrived, we argue that since these objects contain *names* (= pointers) to other objects, a more elegant approach would use a garbage-collected heap. But classical garbage collected heaps have the problem that the allocation routine occasionally calls the garbage collector, which takes an amount of time proportional to the size of the heap to finish. During this hiatus, the heap is unavailable to the rest of the system.

The next chapter presents a new heap management algorithm which works incrementally, by performing a little of the work of garbage collection on every call to the storage allocation routine. In this way, the huge *variance* in the amount of time required to allocate a block of storage is reduced to zero. This algorithm aids in the programming of a system with hard real-time constraints because the time required to allocate an object and access its parts is completely predictable.

Because our heap management algorithm is essentially a real-time simulation of a "list memory" (in the spirit of IPL-V [69] or a LISP machine [29,41,11]) on a "random access memory", we will often use the phrase "list memory" instead of "garbage-collected heap".

3.1 Advantages of List Memory over Random Access Memory

The question arises as to why we go to so much effort to simulate a list memory on an automaton with a random access memory. After all, with a random access memory, one can access any memory cell in the address space in unit time, whereas one must trace lists to access most of the memory cells of a list memory.

The answer is that we rarely use the completely random access ability of the RAM. The actions of a CPU in executing an instruction stream are highly predictable, since most programs consist of lists of instructions with a few conditional and unconditional branches thrown in; in other words, most programs are list structure themselves! The run time systems of higher level languages include stack structured or tree structured variable binding environments which again do not make full use of the random access abilities of the memory.

In fact, the only two constructs that do make essential use of the random access property of the memory are FORTRAN-style arrays of memory cells[1] and hash tables. However, even in applications which use arrays, we often see more structure than a simple one-to-one mapping of indices to memory cells. If the arrays are multidimensional, many systems store them as vectors of addresses to other vectors--i.e. multilevel structures. If the arrays are sparse, they are sometimes stored as doubly-linked list structures or in hash tables, either to speed up processing or reduce storage, or both. Even arrays without such sophisticated structure are usually processed in row or column order, and rarely are accesses made to random array elements. In fact, most arrays which are not scanned linearly are being used to simulate list structure! Thus even array structures, for which random access memory should be ideal, do not normally take advantage of *random* accessing.

---

1. The semantics of an array require that adjacent elements of an array occupy adjacent storage locations so that a probe of a random element in the array takes approximately O(1), regardless of the size of the array.

Hash tables, which simulate an associative memory by interpreting a key--suitably transformed--as an index to memory, make the most important use of the random accessing ability of these memories. However, even this use is limited, since most hash table algorithms do a linear search of the bucket which is chosen by hashing. It is also difficult to extend hash tables, because doing so requires copying and rehashing every element of the table to a new, larger table according to a new hashing function.[2] This brings us to the primary problem of random accessed memory--it is extremely hard to reorganize and re-allocate memory because much data must be physically moved, and this movement is expensive.

List memory, on the other hand, satisfies a *substitution property* which has both a stronger and a weaker form. The stronger form of this property states that any single instance of a list node or atom in a list structure can be replaced by another piece of list structure or an atom, with only a minor, local change to the list memory. This substitution requires only a constant amount of time if the instance to be substituted for is already known and the change is to be permanent.[3] These substitutions furthermore do not affect the access paths to the nodes of the memory which have not been substituted for; hence there is much less need for synchronization among multiple processes making structural changes to a list memory than among multiple processes moving data around in a random access memory.

This substitution property is related to the phrase structure property of higher level languages such as Algol or LISP, where a whole subexpression can also appear in most contexts in which a constant or variable can appear. This feature contrasted with early FORTRAN experience which allowed full expressions in only a few contexts. The free substitution of an expression in contexts where constants or variables are allowed is also called *referential transparency* and is an artifact of the evaluation of expressions in those

---

2. This can be done incrementally, as the next chapter indicates.
3. A whole list can also be substituted for one of its own sublists, thus generating a directed loop in the structure of the list memory.

contexts only for the purpose of the *value* they produce, not the side-effects they cause. (To the extent that languages allow side-effects of expressions, they violate the principle of referential transparency.) This substitution property also operates in *context-free languages*, wherein a non-terminal generates the same sublanguage, regardless of the surrounding context.

The substitution property works in all these systems because they are based on *tree* and *graph* structures rather than on linear *strings* and *vectors*. A tree may sprout new branches from any limb without disturbing the other branches, but inserting new elements in the middle of a string or a vector will affect all accesses to the elements after the inserted part, because they are now further from the beginning (or the end) of the string.

The weaker sense of the substitution property preserves the conceptual idea of subtree replacement, but instead of making a permanent change in the list memory structure, enough of the main tree is copied with the new subtree replacing the old subtree such that the new tree "looks like" the old one, except for the substituted subtree. The conceptual sense of substitution is retained, because each subtree except for the one replaced can still be accessed in the new tree via the same access path that it had in the old one. However, because every node on the access path from the root of the new tree to the substituted subtree is a newly created node, the change is not local and the time to perform the operation is not bounded. However, in most cases the depth of the tree will be only O(log N), where N is the total number of nodes in the tree, so that this type of reorganization is still much cheaper than re-organizing a random access memory, which would require time O(N).

A pleasant result of the use of the substitution property--either in its strong or weak form--is that identical subtrees can be shared because list memory allows an arbitrary directed graph structure. Thus, where the concatenation of *strings* which are represented explicitly requires that the strings be copied into a new area of storage, a list memory allows the representation of a string as the fringe of a tree structure, where some of the subtrees can be shared with representations of other strings. In such a representation, concatenation does not require the copying of the constituents, but requires only the formation of a new

node which points to the two constituent substrings. In actual practise in symbolic manipulation systems [40,76], such shared representations save a great deal of storage, and if processing algorithms know of such sharing, they can sometimes save a great deal of time by considering each shared substructure only once, when it is first encountered by the algorithm, instead of every time it is encountered.

3.2 Allocation Problems of Random Access Storage

Computational complexity theorists have made great strides in the past ten years in identifying and proving certain tasks and problems "hard". While what constitutes a "hard" problem may vary somewhat depending upon your patience and budget, nearly everyone agrees that if the time or space required to compute the answer goes up at least exponentially with the size of the input parameters, then the problem is hard. Now there is a class of problems called *NP-complete* problems which have not yet been proved to require exponential behavior on the standard deterministic serial computer, but for which all existing algorithms are exponential. One of the largest subclasses of the NP-complete class consists of allocation and scheduling problems, which for our purposes refer to storage and time allocation. In fact, almost all allocation and scheduling problems which involve discrete sizes and times are "hard" problems [90,24].

A real-time system requires response delays to stimuli which are guaranteed to be within specified tolerances. The resources required for such a response vary with the current stimulus and the history of preceding stimuli. Two of the most important of those resources are storage cells and processor cycles. Optimal scheduling of *either* storage cells *or* processor cycles alone is an NP-complete packing problem, and scheduling them both together is a two-dimensional packing problem which is surely just as hard. Now if we also require that allocated storage may not be moved between the time it is allocated and the time it is released, then we must also try to minimize *storage fragmentation*, wherein a significant amount of free storage becomes unusable because it is spread throughout the address space in little pieces, none of which is large enough to be usable.

Many designers ignore the fragmentation problem and live with nailed-down storage by giving fixed allocation to all the tables that the system needs and planning very carefully the sizes of the tables. However, this leads to systems which are not robust, which break down when faced with a situation slightly different from that envisioned by the system designers. These systems break down with a message indicating that some obscure table has overflowed and in many cases the problem is uncorrectable because the table sizes cannot be changed. However, even if the system designer wanted to design a fail-soft system--i.e. one which would fail totally only when all resources were uniformly exhausted--he would find it very hard to do so and still stay within the real-time requirements of his application, because of the large amount of copying involved in the reorganization of random access storage.

Enter the list memory and our real-time simulation of it on a random access memory. Using this scheme, the system designer can solve his problems with a much more flexible memory paradigm than the random access memory. He can design his system with a list memory having a conceptually infinite number of cells, which are all interchangeable, and hence only the total number required would matter, not the order in which they were used. Furthermore, so long as the total number of *accessible* cells remains less than the maximum allowed by the memory, he need not worry about the memory becoming fragmented through combinations of allocations and deletions. If his cell requirements grew by a factor of 10 or 10 million, he need not change *one bit* of his program, since there are no addresses stored and hence no address space limitations.[4] With current hardware (real) address spaces growing by approximately one bit per year, he need not worry that his program will become obsolete in only a few years.

---

4. A user program need never know that actual size of a list memory pointer, since the program will never deal with one directly, but only through commands which change the state of a root. Therefore, the program is unaffected by a change in pointer size.

The list memory eliminates the problem of fragmentation and table growth, thus reducing the allocation constraints under which real-time systems must operate. The scheduling of time in these systems remains hard, since real-time systems continue to be at the mercy of their stimuli, but at least we will have given them better control over their own internal storage.

Although we show how a serial computer can do list processing in real time, no current state-of-the art computer is entirely serial. Most have hardware interrupt capabilities and external hardware DMA (direct memory access) I/O devices. DMA devices cause trouble since they ignore the list structure that the system is imposing upon the memory and require that their buffers be nailed down for the duration of the DMA transfer. This lacuna can only be fixed by making the DMA device respect the list structure of the memory.

A system using DMA devices is made most modular by using a separate processor as a memory controller which handles access requests from both the CPU and DMA devices and hence preserves the appearance of the memory as a list memory to all the world. Within the next few years, there will be room on a silicon chip to implement both a controller and a large number of memory cells to create a true "list memory chip". Since non-standard memory chips such as FIFO (first-in, first-out) chips are becoming available, why not truly useful devices like list memory chips?

4. List Processing in Real Time

In this chapter,[1] we present and analyze carefully our method for incremental garbage collection. Although presented here in the terminology of LISP, the algorithm works perfectly well for SIMULA class objects [27,15,2] and CLU cluster objects [61]. More to the point, the algorithm is perfect for the small acquaintance arrays encountered in an actor implementation.

A *real-time* list processing system is one in which the time required by the elementary list operations (e.g. CONS, CAR, CDR, RPLACA, RPLACD, EQ, and ATOM in LISP) is bounded by a (small) constant. Classical implementations of list processing systems lack this property because allocating a list cell from the heap may cause a *garbage collection*, which process requires time proportional to the heap size to finish.

A *real-time* list processing system is presented which continuously reclaims garbage, including directed cycles, while linearizing and compacting the accessible cells into contiguous locations to avoid fragmenting the free storage pool. The program is small and requires no time-sharing interrupts, making it suitable for micro-code. Finally, the system requires the same average time, and not more than twice the space, of a classical non-copying implementation, and those space requirements can be reduced to approximately classical proportions by compact list representation.

Arrays of different sizes, a program stack, and hash linking are simple extensions to our system, and reference counting is found to be inferior for many applications.

---

1. This chapter is essentially the same as the paper "List Processing in Real Time on a Serial Computer" [5].

4.1 Introduction and Previous Work

List processing systems such as LISP [64] have slowly gained popularity over the years in spite of some rather severe handicaps. First, they usually interpreted their programs instead of compiling them, thus increasing their running time by several orders of magnitude. Second, the storage structures used in such systems were inefficient in the use of storage; for example, compiling a program sometimes halved the amount of storage it occupied. Third, processing had to be halted periodically to reclaim storage by a long process known as garbage collection, which laboriously traced and marked every accessible cell so that those inaccessible cells could be recycled.

That such inefficiencies were tolerated for so long is a tribute to the elegance and productivity gained by programming in these languages. These languages freed the programmer from a primary concern: *storage management*. The programmer had only to call CONS (or its equivalent) to obtain a pointer to a fresh storage block; even better, the programmer had only to relinquish all copies of the pointer and the storage block would automatically be reclaimed by the tireless garbage collector. The programmer no longer had to worry about prematurely freeing a block of storage which was still in use by another part of the system.

The first problem was solved with the advent of good compilers [67,88] and new languages such as SIMULA especially designed for efficient compilation [27,15,2]. The second was also solved to some extent by those same compilers because the user programs could be removed from the list storage area and freed from its inefficient constraints on representation.[2] Other techniques such as compact list representation ("CDR-coding") [41,11,22] have been proposed which also offer partial solutions to this problem.

---

2. In many cases, a rarely used program is compiled not to save time in its execution, but to save garbage-collected storage space.

This chapter presents a solution to the third problem of classical list processing techniques and removes that roadblock to their more general use. Using the method given here, a computer could have list processing primitives built in as machine instructions and the programmer would still be assured that each instruction would finish in a reasonable amount of time. For example, the interrupt handler for a keyboard could store its characters on the same kinds of lists--and in the same storage area--as the lists of the main program. Since there would be no long wait for a garbage collection, response time could be guaranteed to be small. Even an operating system could use these primitives to manipulate its burgeoning databases. Business database designers no longer need shy away from pointer-based systems, for fear that their systems will be impacted by a week-long garbage collection! As memory is becoming cheaper[3], even microcomputers could be built having these primitives, so that the prospect of controlling one's kitchen stove with LISP is not so far-fetched.

A *real-time* list processing system has the property that the time required by each of the elementary operations is bounded by a constant independent of the number of cells in use. This property does not guarantee that the constant will be small enough for a particular application on a particular computer, and hence has been called "pseudo-real-time" by some. However, since we are presenting the system independent of a particular computer and application, it is the most that can be said. In all but the most demanding applications, the proper choice of hardware can reduce the constants to acceptable values.

Except where explicitly stated, we will assume the classical Von Neumann serial computer architecture with real memory in this chapter. This model consists of a memory, i.e. a one-dimensional array of words, each of which is large enough to hold (at least) the representation of a non-negative integer which is an index into that array; and a central processing unit, or CPU, which has a small fixed number of registers the size of a word. The CPU can perform most operations on a word in a fixed, bounded amount of time.

---

3. Work is progressing on 10^6 bit chips.

The only operations we require are load, store, add, subtract, test if zero, and perhaps some bit-testing. It is hard to find a computer today without these operations.

As simple as these requirements are, they do exclude virtual memory computers. These machines are interesting because they take advantage of the locality of reference effect, i.e. the non-zero serial correlation of CPU accesses to memory, to reduce the amount of fast memory in a system without greatly increasing the average access time. However, the time required to load a particular word from virtual memory into a CPU register is bounded only by the time to access the slowest memory. Since we are more interested in tight upper bounds, rather than average performance, this class of machines is excluded.

Since the primary list processing language in use today is LISP, and since most of the literature uses the LISP paradigm when discussing these problems, we will continue this tradition and center our discussion around it. Due to its small cells, which consist of 2 pointers apiece, LISP is also a kind of worst case for garbage collection overhead.

There are two fundamental kinds of data in LISP: *list cells* and *atoms*. List cells are ordered pairs consisting of a *car* and a *cdr*, while atoms are indecomposable. ATOM(x) is a predicate which is true if and only if x is an atom (i.e. if and only if x is not a list cell); EQ(x,y) is a predicate which is true if and only if x and y are the same object; CAR(x) and CDR(x) return the car and cdr components of the list cell x, respectively; CONS(x,y) returns a new (not EQ to any other accessible list cell) list cell whose car is initially x and whose cdr is initially y; RPLACA(x,y) and RPLACD(x,y) store y into the car and cdr of x, respectively. We assume here that these seven primitives are the only ones which can access or change the representation of a list cell.

There have been several attempts to tackle the problem of real time list processing. Knuth [57, p. 422] credits Minsky as the first to consider the problem, and sketches a multiprogramming solution in which the garbage collector shares time with the main list processing program. Steele's [80] was the first in a flurry of papers about *multiprocessing* garbage collection which included contributions by Dijkstra [31,32] and Lamport [58,59]. Muller [68] independently detailed the Minsky-Knuth-Steele method, and both he and Wadler [93] analyzed the time and storage required to make it work.

The Minsky-Knuth-Steele-Muller-Wadler (MKSMW) method for real-time garbage collection has two processes running in parallel. The list processor process is called the *mutator* while the garbage collector is called the *collector* (these terms are due to Dijkstra [31]). The mutator executes the user's program while the collector performs garbage collection, over and over again. The collector has three phases: *mark*, *sweep*, and *relocate*. During the mark phase, all accessible storage is marked as such, and any inaccessible storage is picked up during the sweep phase. The relocate phase relocates accessible cells in such a way as to minimize the address space required. Since the mutator continues running while the mark and relocate phases proceed, the free list must be long enough to keep the mutator from starvation. During the sweep phase, cells must be added to the free list *faster* than they can be taken off, on the average, else the net gain in cells from that garbage collection cycle would be negative.

Muller [68] and Wadler [93] have studied the behavior of this algorithm under equilibrium conditions (when a cell is let go for every cell CONS'ed, and when the rates of cell use by the mutator, and of marking, sweeping, and relocating by the collector, are all constant). If we let m be the ratio of the rate of CONS'ing to that of marking, s be the ratio of the rate of CONS'ing to that of sweeping, and r be the ratio of the rate of CONS'ing to that of relocating, then we can derive estimates of the size of storage needed to support an accessible population of N cells under equilibrium conditions.[4] Using these assumptions, we derive:

$$\text{Maximum MKSMW Storage Required} \le N \frac{m+(m+1)(r+1)}{1-s(r+1)} + \text{size of collector stack}$$

We note that r=0 if there is no relocation (i.e. it happens instantaneously), in which case we have the simpler expression:

---

4. Of course s<1, or else the storage required is infinite.

$$\text{Maximum MKSMW Storage Required} \leq N \frac{1+2m}{1-s} + \text{size of collector stack}$$

The collector stack seems to require depth N to handle the worst case lists that can arise, but each position on the stack need only hold one pointer. Since a LISP cell is two pointers, the collector stack space requirement is .5N. Thus, we arrive at the inequality:

$$\text{Maximum MKSMW Storage Required} \leq N \frac{1.5+2m-.5s}{1-s}$$

These estimates become bounds for non-equilibrium situations so long as the ratios of the rate of CONS'ing to the rates of marking, sweeping, and relocating are constant. In other words, we relativize the rates of marking, sweeping, and relocating with respect to a cons-counter rather than a clock.

The Dijkstra-Lamport (DL) method [31,32,58,59] also has the mutator and collector running in parallel, but the collector uses no stack. It marks by scanning all of storage for a mark bit it can propagate to the marked cell's offspring. This simple method of garbage collection was considered because their main concern was *proving* that the collector actually collected only and all garbage. Due to its inefficiency, we will not consider the storage requirements of this method.

Both the MKSMW and the DL methods have the drawback that they are parallel algorithms and as a result are incredibly hard to analyze and prove correct. By contrast, the method we present is serial, making analyses and proofs easy.

4.2 The Method

Our method is based on the Minsky garbage collection algorithm [66], used by Fenichel and Yochelson in an early Multics LISP [34], elegantly refined by Cheney [20], and applied by Arnborg to SIMULA [2]. This method divides the list space into two *semispaces*. During the execution of the user program, all list cells are located in one of the semispaces. When garbage collection is invoked, all accessible cells are traced, and instead

of simply being marked, they are moved to the other semispace. A *forwarding address* is left at the old location, and whenever an edge is traced which points to a cell containing a forwarding address, the edge is updated to reflect the move. The end of tracing occurs when all accessible cells have been moved into the "to" semispace (*tospace*) and all edges have been updated. Since the tospace now contains all accessible cells and the "from" semispace (*fromspace*) contains only garbage, the collection is done and the computation can proceed with CONS now allocating cells in the former fromspace.

This method is simple and elegant because 1) it requires only one pass instead of three to both collect and compact, and 2) it requires no collector stack. The stack is avoided through the use of two pointers, B and S. B points to the first free word (the *bottom*) of the free area, which is always in the tospace. B is incremented by COPY, which transfers old cells from the fromspace to the bottom of the free area, and by CONS, which allocates new cells. S *scans* the cells in tospace which have been moved, and updates them by moving the cells they point to. S is initialized to point to the beginning of tospace at every flip of the semispaces and is incremented when the cell it points to has been updated. At all times, then, the cells between S and B have been moved, but their cars and cdrs have not been updated. Thus when S=B all accessible cells have been moved into tospace and their outgoing pointers have been updated. This method of pointer updating is equivalent to using a queue instead of a stack for marking, and therefore traces a spanning tree of the accessible cells in breadth-first order.

Figure 13 shows a diagram of this algorithm in operation.

Besides solving the compaction problem for classical LISP, the Minsky-Fenichel-Yochelson-Cheney-Arnborg (MFYCA) method allows simple extensions to handle non-uniformly sized arrays and CDR-coding because free storage is kept in one large block. Allocation is therefore trivial; to allocate a block of size n, one simply adds n to the "free space pointer".

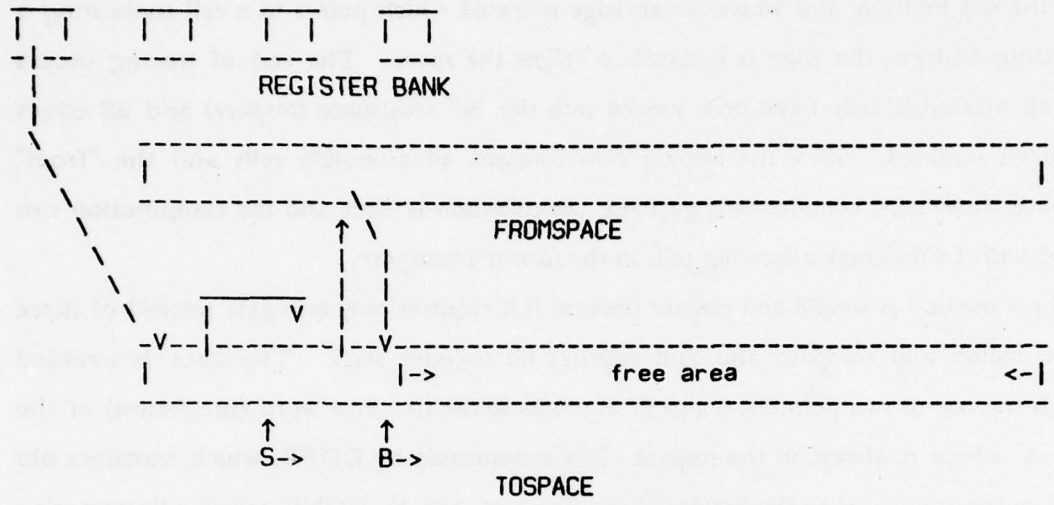
Copying garbage collectors have been dismissed by many as requiring too much storage for practical use (because they appear to use twice as much as classical LISP), but we shall see that perhaps this judgment was premature.

AD-A053 328 MASSACHUSETTS INST OF TECH CAMBRIDGE LAB FOR COMPUTE--ETC F/G 9/4
ACTOR SYSTEMS FOR REAL-TIME COMPUTATION.(U)
MAR 78 H G BAKER N00014-75-C-0522
UNCLASSIFIED MIT/LCS/TR-197 NL

2 OF 2
AD
A053328

END
DATE
FILMED
6-78
DDC

Fig. 13. The Cheney Algorithm



We present the MFYCA algorithm here as Figure 14 in pseudo-Algol-BCPL notation. The notation "$\alpha[\beta]$" means the contents of the word whose address is the value of α plus the value of β, i.e. the contents of $\alpha+\beta$. If it appears on the left hand side of ":=", those contents are to be changed. Thus, $p[i]$ refers to the i-th component of the vector pointed to by p. The function *size(p)* returns the size of the array pointed to by p. The notation "α & β" is similar to the notation "$\alpha;\beta$" in that α and β are executed in order; however, "α & β" returns the value of α rather than the value of β. Thus, ";" and "&" are the duals of one another: "$\alpha_1;\alpha_2;\ldots;\alpha_n$" returns the *last* value (that of α_n) whereas "$\alpha_1\&\alpha_2\&\ldots\&\alpha_n$" returns the *first* value (that of α_1).

Our conventions are these: the user program has a bank of NR registers $R[1],\ldots,R[NR]$. *The user program may not "squirrel away" pointers outside of the bank R during a call to CONS* because such pointers would become obsolete if garbage collection were to occur. (We will show later how to deal with a user program stack in such a way that the real-time properties of our system are not violated.) Pointers either are *atoms* or refer to *cons cells* in fromspace or tospace. A cons cell c is represented by a 2-vector of pointers: $car(c)=c[0]$, $cdr(c)=c[1]$. FLIP, FROMSPACE and TOSPACE are implementation

dependent routines. FLIP interchanges the roles of fromspace and tospace by causing CONS and COPY to allocate in the other semispace and the predicates FROMSPACE and TOSPACE to exchange roles. FLIP also has the responsibility of determining when the new tospace is too small to hold everything from the fromspace plus the newly CONS'ed cells. Before flipping, it checks if *size*(fromspace) is less than (1+m)[*size*(tospace)-(T-B)], where m is a constant parameter and T is the top of tospace. If FLIP finds that fromspace (the new tospace) is too small, either it must extend the space, or the system may later stop with a "memory overflow" indication.

In order to convert MFYCA into a real-time algorithm, we force the mark ratio m to be constant by changing CONS so that it does k iterations of the garbage collection loop before performing each allocation. But this means that both semispaces contain accessible cells at almost all times. In order to simplify the algorithm and the proof, *we trick the user program into believing that garbage collection ran and finished at the time of the last flip;* i.e. we assert that, as before, the user program sees addresses only in tospace.

Some slight effort must be made to keep up this appearance. When the semispaces are interchanged, all the user program registers must be updated immediately to point to tospace. This gives the collector a head start on the mutator. Since the only operations that might violate our assertion are CAR and CDR, we make sure that CAR and CDR cause forwarding addresses to be followed, and cells to be moved, when necessary. This ensures that the mutator cannot pass the collector. It turns out that preserving our assertion is much simpler than preserving the corresponding assertions of DL [31,32,58,59]. In particular, RPLACA and RPLACD do not cause any trouble at all!

There is another problem caused by interleaving garbage collection with normal list processing: the new cells that CONS creates will be interleaved with those moved, thereby diluting the moved cells which must be traced by CONS. Of course, new cells have their cars and cdrs already in tospace and therefore do not need to be traced. We avoid this waste of trace effort through the use of the pointer T, which points to the *top* of the free area, where we will allocate all new cells.

Fig. 14. The Minsky-Fenichel-Yochelson-Cheney-Arnborg Garbage Collector

```
pointer B;                              % Bottom; points to bottom of free area. %
pointer S;                              % Scan; points to first untraced cell.  %
pointer T;                              % Top; points to top of tospace.  %
                                        % Assertions: S ≤ B ≤ T and T-B is even.%
pointer procedure CONS(x,y) =           % Allocate the list cell (x . y). %
  begin
    if B=T                              % If there is no more free space, %
       then                             %  collect all the garbage.  %
         begin                          % This block is the "garbage collector". %
           flip();                      % Interchange semispaces. %
           for i = 1 to NR              % Update all user registers. %
               do R[i]:=move(R[i]);
           x:=move(x); y:=move(y);      % Update our arguments.  %
           while S<B                    % Trace all accessible cells. %
                 do begin
                      S[0]:=move(S[0]); % Update the car and cdr. %
                      S[1]:=move(S[1]);
                      S := S+2          % Point to next untraced cell.  %
                    end
         end;
    if B≥T then error;                  % Memory is full.  %
    B[0] := x; B[1] := y;               % Create new cell at bottom of free area.%
    B & (B := B+2)                      % Return the current value of B %
  end;                                  %  after stepping it to next cell. %

pointer procedure CAR(x) = x[0];        % A cell consists of 2 words; %

pointer procedure CDR(x) = x[1];        %  car is 1st; cdr is 2nd.  %

procedure RPLACA(x,y) = x[0] := y;      % car(x) := y %

procedure RPLACD(x,y) = x[1] := y;      % cdr(x) := y %

boolean procedure EQ(x,y) = x=y;        % Are x,y are the same object? %

boolean procedure ATOM(x) =             % Is x an atom? %
  not tospace(x);

pointer procedure MOVE(p) =             % Move p if needed; return new address.%
  if not fromspace(p)                   % We only need to move old ones. %
     then p                             % This happens a lot. %
     else begin
            if not tospace(p[0])        % We must move p. %
               then p[0] := copy(p);    % Copy it into the bottom of free area. %
            p[0]                        % Leave and return forwarding address.  %
          end;

pointer procedure COPY(p) =             % Create a copy of a cell. %
```

```
  begin                                  % Allocate space at bottom of free area.%
    if B≥T then error;                   % Memory full? %
    B[0] := p[0]; B[1] := p[1];          % Each cell requires 2 words %
    B & (B := B+2)                       % Return the current value of B %
  end;                                   %  after moving it to next cell. %
% TOSPACE, FROMSPACE test whether a pointer is in that semispace.  %
```

Figure 15 shows a diagram of our incremental method in operation, while figure 16 presents the code for our real-time list processing system.

The time required by all of the elementary list operations in this algorithm, with the exception of CONS, can easily be seen to be bounded by a constant because they are straight-line programs composed from primitives which are bounded by constants. CONS is also bounded by a constant because the number of mutator registers is a (small) fixed number (e.g. 16), and the parameter k is fixed. In principle, given the number of registers and the parameter k, the two loops in CONS could be expanded into straight-line code; hence the time it requires is also bounded by a constant.

The proof that the incremental collector eventually moves all accessible cells to tospace is an easy induction. Upon system initialization there are no accessible cells, hence none in tospace, and so we have a correct basis. Suppose that at some point in the computation we have just switched semispaces so that tospace is empty. Suppose further that there are N accessible cells in fromspace which must be moved to tospace. Now, every cell which was accessible at the time of flipping eventually gets moved when it is traced, unless lost through RPLACA and RPLACD, and as a result appears between S and B. Furthermore, a cell is

Fig. 15. The Serial Real-Time Method

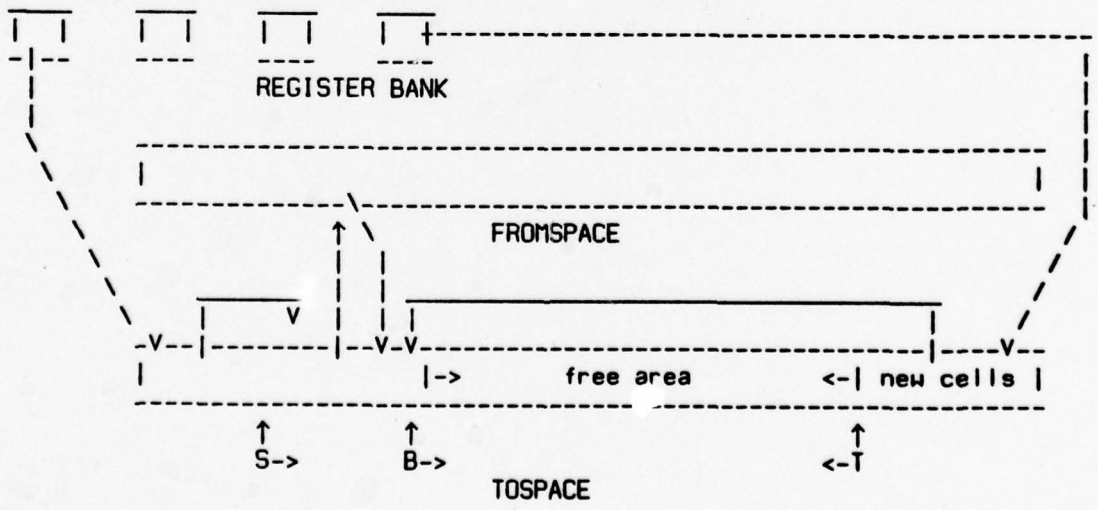


Fig. 16. The Serial Real-Time List Processing System

```
integer k;                                  % Global trace ratio parameter: the
                                              number of cells to trace per cons.%
pointer T;                                  % Top; Points to top of free area. %

pointer procedure CONS(x,y) =               % Do some collection,
                                              then allocate (x . y). %  .
  begin
    if B=T                                  % Check if free area is empty. %
      then begin                            % Switch semispaces.  Memory is full %
             if S<B then error;             %  if tracing is not finished. %
             flip();                        % Flip semispaces. %
             for i = 1 to NR
                 do R[i]:=move(R[i]);       % Update user registers %
             x:=move(x); y:=move(y)         %  and our arguments. %
           end;
    for i = 1 to k while S<B                % Do k iterations of gc. %
        do begin
             S[0]:=move(S[0]);              % Update car and cdr. %
             S[1]:=move(S[1]);
             S := S+2                       % Go on to next untraced cell.  %
           end;
    if B=T then error;
    T := T-2;                               % Actually create the cell. %
    T[0] := x; T[1] := y;                   % Move in car and cdr.  %
    T                                       % Return address of new cell.  %
  end;

pointer procedure CAR(x) =                  % Move, update and return x[0]. %
  x[0] := move(x[0]);

pointer procedure CDR(x) =                  % Move, update and return x[1]. %
  x[1] := move(x[1]);

%       Procedures not redefined here are as before.                  %
```

---

moved only once, because when it is moved it leaves behind a forwarding address which prevents it from being moved again. When the pointer S reaches a cell, its edges are traced--i.e. the cells they point to are moved, if necessary. Finally, only cells which have been moved appear between S and B. Therefore, the number of those accessible, unmoved cells in fromspace decreases monotonically, eventually resulting in no accessible, unmoved cells in fromspace. At this point, the collector is done and can interchange the two semispaces.

It should be easy to see why the other list operations cannot adversely affect the progress of the collector. A CAR or CDR can move a cell before the collector has traced it, but since moving it increases B but not S, it will be traced later. RPLACA and RPLACD can affect connectivity, but since all of their arguments are already in tospace, they have already been moved and may or may not have been traced. Consider RPLACA(p,q). Suppose that p has been traced and q has not. But since q has been moved but not traced, it must be between S and B and will not be missed. Suppose, on the other hand, that q has been traced and p has not. Then when p is traced, the old CAR of p will not be traced. But this is all right, because it may no longer be accessible. If it still is the target of an edge from some accessible cell, then it either already has, or will be, traced through that edge. Finally, if either both p and q have been traced or both have not been, there is obviously no problem.

This algorithm can also be proved correct by the methods of DL [31,32,58,59], because this particular sequence of interleaving collection with mutation is only one of the legal execution sequences of the DL algorithm on a serial machine. Therefore, if the DL algorithm is correct, then so is this one. The correspondence is this: *white* nodes are those which reside in fromspace, i.e. those which have not yet been moved; *grey* nodes are those which have been moved but have not yet been traced, i.e. those between S and B; and *black* nodes are those which have been moved and traced, and those which have been allocated directly in tospace (cells below S or above T). Then the assertions are:

A) each node will only darken monotonically;

B) no edge will ever point from a black node to a white one; and

C) the user program sees only grey or black nodes.

We can now see why the burden is on CAR and CDR rather than RPLACA and RPLACD--the latter will not violate B so long as the former do not violate C. Using these assertions, we see that the mutator and the mark phase of the collector are essentially doing the same thing: tracing accessible cells. The difference is that the collector goes about it systematically whereas the mutator wanders. Thus, only the collector knows for sure when all the cells in fromspace have been traced so that the two semispaces can be interchanged.

Assertion C also allows CAR and CDR to update a cell in which a pointer to fromspace is found, thus reducing pointer-chasing for cells which are accessed more than once.

We must now analyze the storage required by this algorithm. Suppose that at some flip of the semispaces there are N accessible nodes. Then the collector will not have to move or trace any more than N cells. If it traces (makes black) exactly k cells per CONS, then when the collector has finished, the new semispace will contain $\leq N+N/k = N(1+m)$ cells, where we let $m=1/k$. If only N of these are accessible, as in equilibrium conditions, then the next cycle of the collector will copy those N cells back to the first semispace, while performing Nm CONS'es. Hence, we have the inequality:

Maximum SRT Storage Required $\leq N(2+2m) = N(2+2/k)$

Therefore, for a program which has a maximum cell requirement of N cells operating on a fixed-size real memory of 2M cells, the parameter k must be greater than N/(M-N) to guarantee that tracing is finished before every flip.

If we compare the bound for our algorithm with the bound for MKSMW, using the unlikely assumption that sweeping and relocation take no time (s=r=0), we find that they are quite similar in storage requirements.

Maximum MKSMW Storage Required $< N(1.5+2m)$
Maximum SRT Storage Required $\leq N(2+2m)$

If m=1 (which corresponds to one collector iteration per CONS), the two algorithms differ by only 1 part in 8, which is insignificant given the gross assumptions we have made about MKSMW's sweeping and relocation speeds. It is not likely that the storage requirements of a MKSMW-type algorithm can be significantly improved because it cannot take advantage of techniques like stack threading or CDR-coding. Stack threading cannot be done, because accessible cells have both their car and cdr in use.[5] CDR-coding using MKSMW is very awkward because CONS must *search* for a free cell of the proper size and location before allocating a cell, since the free space is fragmented. On the other hand, our

---

5. The Deutsch-Schorr-Waite collector [57, p. 417-418] "threads" the stack but temporarily reverses the list structure, thus locking out the mutator for the duration.

algorithm can be easily modified to use CDR-coding and thereby reduce storage requirements to approximately N(1+m).

4.3 The Parameter m (= 1/k)

If k is a positive integer, then the parameter m (=1/k) will lie in the interval $0<m\leq 1$. Therefore, the factor of 1+m in our bounds must lie between 1 and 2. This means that the storage requirements for our method can be adjusted by varying k, but they will not vary by more than a factor of 2 (so long as k is integral). Now, the time to execute CONS is proportional to k+c, for some suitable constant c. Therefore, one can trade off storage for CONS speed, but only within this limited range. Furthermore, as k rises above 4 the storage savings become insignificant; e.g. doubling k to 8 yields a storage savings of only 10%, yet almost doubles CONS time. Of course, if storage is limited and response time need not be fast, larger k's might be acceptable.

If the method is used for the management of a large database residing on secondary storage, k could be made a positive rational number less than 1, on the average. For example, to achieve an average k=1/3 (m=3), one could have CONS perform an iteration of the collector only every *third* time it was called. The result of this would double the storage required (m+1=4), but would reduce the average CONS time by almost 2/3. Of course, the *worst case* time performance for CONS would still be the same as if k were 1.

This improvement is significant because each iteration of the collector traces all the pointers of one record. This requires retrieving that record, updating all of its pointers by moving records if necessary, and then rewriting the record. If there are t pointers to be updated, then $t+1$ records must be read and written. This sounds like a lot of work, but this much work is done only when a record is created; if there are no record creations, then with the exception of the first access of a record via a pointer stored in another record, the accessing and updating functions will be as fast as on any other file management scheme. Therefore, since secondary storage is usually cheap but slow, choosing k<1 in a file management system allows us to trade off storage space against average record creation

time.

With a little more effort, k can even be made *variable* in our method, thus allowing a program to dynamically optimize its space-time tradeoff. For example, in a database management system a program might set k=0 during an initial *load* of the database because it knows that even though there are many records being created, none are being let go, and therefore the continual copying of the collector will achieve no compaction. The function READ in LISP might want to exercise the same prerogative, for the same reason. Of course, any reduction of k should not take effect until the next flip, to avoid running out of storage before then.

4.4 A User Program Stack

If the user program utilizes its own stack as well as a bank of registers, the stack may (in theory) grow to an unbounded size and therefore cannot be wholly updated when the semispaces are flipped and still preserve a constant bound on the time for CONS. This problem may be trivially solved by simulating the stack in the heap (i.e. PUSH(x) ≡ CONS(x, stack) and POP() ≡ CDR(stack)); this simulation will satisfy the bounded-time constraints of classical stack manipulation. However, this simulation has the unfortunate property that accessing items on the stack requires time proportional to their distance from the top.

In order to maintain constant access time to elements deep in the stack, we keep stack-like allocation and deallocation strategies but perform the tracing of the stack in an incremental manner. We first fix the stack accessing routines so that the user program never sees pointers in fromspace. This change requires that the MOVE routine must be applied to any pointers which are picked up from the user stack. We must then change CONS to save the user stack pointer when the semispaces are flipped so that it knows which stack locations must be traced. Finally, the user stack POP routine must keep this saved pointer current to avoid tracing locations which are no longer on the user stack [68].

The only remaining question is how many stack locations are to be traced at every CONS. To guarantee that stack tracing will be finished before the next flip, we must allocate the stack tracing ratio k' (the number of stack locations traced per CONS) so that the ratio k'/k is the same as the ratio of stack locations in use to cons cells in use. We recompute k' at each flip, because the "in use" statistics are available then. Due to this computation, a constant bound on the time for CONS exists only if the ratio of stack size to heap size is bounded, and is proportional to that ratio.

Figure 17 exhibits these changes.

Barbara Liskov [62] has suggested tracing the user stack from the bottom instead of from the top, as we have done here. The rationale behind this is that many of the pointers at the top of the stack will have been discarded by the time the collector gets there, and those discarded pointers may never be traced. Space requirements may be slightly reduced as a result, since some garbage may be detected earlier. However, since the mutator must continue checking (and possibly tracing) every pointer it accesses from the stack, the change results in only a marginal improvement in time.

The complexity involved in this conversion is essentially that necessary to make the serial real-time method work for several different *spaces* [67]. In such a system, each space is a contiguous area in the address space disjoint from the other spaces, and has its own representation conventions and allocation (and deallocation) strategies. The system of this section thus has two spaces, the heap and the user stack, which must be managed by cooperating routines.

4.5 CDR-Coding (Compact List Representation)

In this section, we discuss the interaction of our algorithm with a partial solution to the second big problem with list structures: their inefficient use of storage. Whereas a list of 5 elements in a language like Fortran or APL would require only a 5 element array, such a list in LISP requires 5 cells having two pointers apiece. So-called "CDR-coding" [41,11,22] can reduce the storage cost of LISP lists by as much as 50%. The idea is simple: memory is

Fig. 17. Real-Time System with User Stack

```
%       The user stack resides in the array "ustk" and grows upward from
        "ustk[0]".  The global variable "SP" is the user stack pointer and
        points to the current top of the user stack.  The global variable "SS"
        scans the user stack and points to the highest stack level which
        has not yet been traced by the collector.  %

integer SP init(0);                         % User stack pointer. %
integer SS init(0);                         % User stack scanner. %

procedure USER_PUSH(x) ≡                    % Push x onto user stack. %
  begin                                     % Note: x will not be in fromspace. %
    SP := SP+1;
    ustk[SP] := x
  end;

pointer procedure USER_POP() ≡              % Pop top value from user stack. %
  move(ustk[SP]) &                          % Move value if necessary; %
  begin
    SP := SP-1;                             %  then update stack pointer. %
    SS := min(SS,SP)                        % Keep stack scanner current. %
  end;

pointer procedure USER_GET(n) ≡             % Get n'th element from top of stack. %
  ustk[SP-n] := move(ustk[SP-n]);           % Move and update if necessary. %

pointer procedure CONS(x,y) ≡               % Collect some, then allocate (x . y). %
  begin
    if B=T                                  % Check if free area is empty. %
      then begin                            % Interchange semispaces.  %
             if SS>0 or S<B                 % Check for memory overflow.  %
               then error;
             N := flip();                   % Set N to number of cells in use.  %
             SS := SP;                      % Start stack scan at top of stack. %
             k' := ceil(k*SS/N);            % Calculate stack trace effort.  %
             for i = 1 to NR
                 do R[i]:=move(R[i]);       % Update user registers %
             x:=move(x); y:=move(y)         %  and our arguments. %
           end;
    for i = 1 to k' while SS>0              % Move k' user stack elements and %
        do begin                            %  update scan pointer. %
             ustk[SS]:=move(ustk[SS]);
             SS := SS-1
           end;
    for i = 1 to k while S<B                % Do k iterations of gc. %
        do begin
             S[0] := move(S[0]);            % Trace & update car, cdr.  %
             S[1] := move(S[1]);
             S := S+2
```

```
        end;
    if B=T then error;
    T := T-2;                          % Actually create the cell. %
    T[0] := x; T[1] := y;              % Install car and cdr. %
    T                                  % Return address of copied cell. %
end;
```

---

divided up into equal-sized chunks called Q's. Each Q is big enough to hold 2 bits plus a pointer *p* to another Q. The 2 bits are decoded via the following table:

00 - NORMAL; CAR of this node is *p*; CDR is *in* the following Q.

01 - NIL; CAR of this node is *p*; CDR is NIL.

10 - NEXT; CAR of this node is *p*; CDR *is* the following Q.

11 - EXTENDED; The cell extension located at *p* holds the car and cdr for this node.[6]

CDR-coding can reduce by 50% the storage requirements of a group of cells for which CDR is a 1-1 function whose range excludes non-nil atoms. This is a non-trivial saving, as all "dot-less" s-expressions read in by the LISP reader have these properties. In fact, Clark and Green [22] found that after linearization 98% of the non-NIL cdrs in several large LISP programs referred to the following cell. These savings are due to the fact that CDR-coding takes advantage of the implicit linear ordering of addresses in address space.

What implications does this coding scheme have for the elementary list operations of LISP? Most operations must dispatch on the CDR-code to compute their results, and RPLACD needs special handling. Consider RPLACD(p,q). If p has a CDR code of NIL or NEXT, then it must be changed to EXTENDED, and the result of CONS(CAR(p),q) placed in p.[7]

The number of memory references in the elementary operations has been minimized by making the following policies [42]:

---

6. These conventions are slightly different from those of [41,11].

7. We note in this context that if RPLACD is commonly used to destructively reverse a list--e.g. by LISP's "NREVERSE"--the system could also have a "PREVIOUS" CDR-code so that RPLACD need not call CONS so often.

1) every EXTENDED cell has a NORMAL extension;

2) the user program will never see a pointer to the extension of an EXTENDED cell; and

3) when COPY copies an EXTENDED cell, it reconstitutes it without an extension.

CONS, CAR, CDR, RPLACA and RPLACD must be changed to preserve these assertions, but EQ and ATOM require no changes from their non-CDR-coded versions. Since an EXTENDED cell cannot point to another EXTENDED cell, the forwarding of EXTENDED pointers need not be iterated. These policies seem to minimize memory references because each cell has a constant (between flips) canonical address, thereby avoiding normalization [80] by every primitive list operation.

CDR-coding requires a compacting, linearizing garbage collector if it is to keep allocation simple (because it uses two different cell sizes) and take full advantage of the sequential coding efficiency. The MFYCA algorithm presented above compacts, but does not linearize cdrs due to its breadth-first trace order. However, the trace order of a MFYCA collector can be easily modified at the cost of an additional pointer, PB. PB keeps track of the previous value of B (i.e. PB points to the last cell copied), so that tracing the cdr of the cell at PB will copy its successor into the next consecutive location (B), thus copying whole lists into successive contiguous locations.

The meaning of the scan pointer S is then changed slightly so that it points to the next *word* which must be updated rather than the next *cell*. Finally, the trace routine is modified so that tracing the cdr of PB has priority over tracing the edge at S and the condition on the trace loop is modified to amortize both the copying effort (measured by movements of B) and the tracing effort (measured by movements of S) over all the CONS'es. These modifications do not result in a depth-first trace order, but they do result in cdr-chains being traced to the end, with few interruptions. Thus an MFYCA collector can minimize the amount of memory needed by CDR-coded lists.

Figure 18 presents a real-time list system which utilizes CDR-coding.

The size of the tospace needed for CDR-coding is (1+m) times the amount of space actually used in fromspace. With a coding efficiency improvement of e over the classical storage of LISP cells, and under equilibrium conditions, we have the inequality:

Maximum SRTC Storage Required $\leq$ Ne(2+2m)

Since we have claimed that e$\approx$.5, we get the following estimate:

SRTC Storage Required $\approx$ N(1+m) (!)

But this latter expression is less than the bound computed for MKSMW. Thus, CDR-coding has given us back the factor of 2 that the copying garbage collector took away.

The real-time properties of our algorithm have not been affected in the least by CDR-coding; in fact, good microcode might be able to process CDR-coded lists faster than normal lists since fewer references to main memory are needed.

CDR-coding is not the final answer to the coding efficiency problems of list storage, because far more compact codes can be devised to store LISP's s-expressions. For example, both the car and cdr of a cell could be coded by relative offsets rather than full pointers [22]. However, a more compact code would represent some cells in so few bits that the pointer we need for a forwarding address would not fit, rendering our scheme unworkable. Part of the problem is inherent in LISP's small cell size; small arrays can perform much better.

4.6 Vectors and Arrays

Arrays can be included quite easily into our framework of incremental garbage collection by simply enclosing certain parts of the collector program in loops which iterate through all the pointers in the array, not just the first and second. The convergence of the method with regard to storage space can also be proved and bounds derived. However, the method can no longer claim to be *real-time* because neither the time taken by the array allocation function (ARRAY-CONS) nor the time taken by the array element accessing function is bounded by a constant. This unbounded behavior has two sources: copying an

Fig. 18. Real-Time System with CDR-Coding

```
pointer S;                                  % Next cell whose car needs tracing. %
pointer PB;                                 % Pointer to previous value of B. %
pointer L,H;                                % Low and high limits of tospace. %
                                            % Assertion: L ≤ S ≤ PB ≤ B ≤ T ≤ H.%
pointer procedure CONS(x,y) =               % Create a new cell in tospace with %
  begin                                     %  car of x and cdr of y. %
    if T-B<2                                % Flip when free area is exhausted. %
    then begin                              % This part is the same as usual. %
           if S<B then error;               % Copying is not done; memory overflow!%
           flip();                          % Interchange semispaces. %
           for i = 1 to NR
           do R[i]:=move(R[i]);             % Update user registers. %
           x:=move(x); y:=move(y)           % Update our arguments. %
         end;
    while (S+B)/2-L < k*(H-T-2) and S<B % Trace and copy a measured amount.%
    do if PB<B                              % Extend current list, if possible. %
       then PB := (B & CDR(PB));            % CDR will trace this edge for us. %
       else begin
              S[0]:=move(S[0]);             % Update this edge. %
              S := S+1                      % Step S over this cell. %
            end;
    if B=T then error;                      % Check for memory overflow. %
    T := T-1;                               % Create new cell at top of free area.%
    if y=nil
    then code(T) := "NIL"                   % If y is special case, %
    else if y=T+1
         then code(T) := "NEXT"             %  then create a short cell %
         else begin                         %  with appropriate cdr-code. %
                if B=T then error;          % Otherwise, create a normal cell. %
                T := T-1;                   % Need more space for the cdr. %
                code(T) := "NORMAL";        % Set in "NORMAL" cdr-code. %
                T[1] := y                   % Set in the cdr. %
              end;
    T[0] := x;                              % Set the car in the new cell. %
    T                                       % Return the new cell. %
  end;

pointer procedure CAR(x) =                  % CAR must move cell it uncovered. %
  brplaca(x, move(bcar(x)));                % Update this edge. %

procedure RPLACA(x,y) =                     % x[0] := y.  May require subtlety. %
  brplaca(x,y);

pointer procedure BCAR(x) =                 % Basic car; dispatch on CDR-code. %
  if code(x)="EXTENDED"                     % Type "EXTENDED" means %
     then (x[0])[0]                         %  indirect car. %
     else x[0];                             % All other types have normal cars. %
```

```
pointer procedure BRPLACA(p,q) =
  if code(p)="EXTENDED"
     then (p[0])[0] := q
     else p[0] := q;

pointer procedure CDR(x) =
  brplacd(x, move(bcdr(x)));

procedure RPLACD(x,y) =
  begin
    if code(x)="NIL" or
       code(x)="NEXT"
       then
         begin pointer p;
           p := CONS(CAR(x),"DUMMY");
           x := move(x); y := move(y);
           x[0] := p;
           code(x) := "EXTENDED"
         end;
    brplacd(x,y)
  end;

pointer procedure BCDR(x) =
  if code(x)="NORMAL" then x[1]
  else if code(x)="NIL" then nil
  else if code(x)="NEXT" then x+1
  else (x[0])[1];


pointer procedure BRPLACD(p,q) =
  if code(p)="EXTENDED"
     then (p[0])[1] := q
  else if code(p)="NORMAL"
          then p[1] := q
  else q;

integer procedure SIZE(p) =
  if code(p)="NORMAL"
     then 2 else 1;

pointer procedure COPY(p) =
  begin
    if PB=B-2 and bcdr(PB)=p
       then begin
              code(PB) := "NEXT";
              B := B-1
            end;
    if bcdr(p)=nil
          then code(B) := "NIL";
```

```
% Basic rplaca; dispatch on CDR-code. %
% If extended cell, clobber indirectly. %

% All others have normal car. %

% CDR moves uncovered cell, but updates %
%  only if still possible after move. %

% x[1] := y.  May require brute force. %

% Test for screw cases. %
% Cannot have code(x)="EXTENDED". %

% Extend the cell x. %
% Construct guaranteed NORMAL cell. %
% Update arguments in case CONS flipped. %
% Leave forwarding address in old cell. %
% The old cell has now been extended. %

% Finally replace the cdr. %


% Basic cdr; dispatch on CDR-code. %
% NORMAL cells have a second word. %
% Interpret NIL CDR-code. %
% Interpret NEXT CDR-code. %
% EXTENDED cells point to NORMAL
   cells. %


% Handle easy cases of RPLACD. %
% We have extended cell; %
%  clobber the NORMAL indirect. %
% The easiest case of all. %

% In all cases, return q as value. %

% Find the size of p from its CDR-code. %
% "NIL", "NEXT", and "EXTENDED"
   all have size(p)=1. %

% Copy the cell p; append to current %
%  train if possible. %
% See if we can hop this NEXT train. %

% Convert NORMAL cell to NEXT cell. %
% Reuse extra space now available. %

% Create a NIL cell, if appropriate. %
```

```
        else code(B) := "NORMAL";      % Otherwise, all cells are NORMAL. %
    B[0] := bcar(p);                   % Copy over car; %
    brplacd(B,bcdr(p));                %  and cdr too, if necessary. %
    PB := B;                           % PB is end of current NEXT train. %
    B := B+size(B);                    % Step B over newly copied cell, %
    if B>T then error;                 %  check for memory overflow, %
    PB                                 %  and return pointer to new copy. %
  end;
% Procedures not redefined here are as before. %
```

array and tracing all its pointers both require time proportional to the length of the array. Therefore, if these operations are included in a computer as non-interruptable primitive instructions, hard interrupt response time bounds for that computer will not exist. However, an arbitrary bound (say 10) placed on the size of all arrays by either the system or the programmer, allows such bounds to be derived.

Guy Steele [83] has devised a scheme which overcomes some of these problems. He gives each vector a special link word which holds either a forwarding pointer (for vectors in fromspace which have been partially moved), a backward link (for incomplete vectors in tospace), or NIL (for complete vectors). MOVE no longer copies the whole array, but only allocates space and installs the forward and backward links. Any reference to an element of a moved but incompletely updated vector will follow the backward link to the fromspace and access the corresponding element there. When the scan pointer in the tospace encounters such a vector, its elements are incrementally updated by applying MOVE to the corresponding elements of its old self; after the new one is complete, its link is set to NIL. Element accesses to incomplete vectors compare the scan pointer to the element address; access is made to the old (new) vector if the scan pointer is less (greater or equal). Tracing and updating exactly kn vector elements (not necessarily all from the same vector) upon every allocation of a vector of length n guarantees convergence.

Steele's scheme has the following properties: the time for referencing an element of any cell or vector is bounded by a constant while the time to allocate a new object of size n is bounded by $c_1 kn + c_2$, for some constants c_1 and c_2. Hence, a sequence of list and vector operations can be given tight time bounds.

4.7 Hash Tables and Hash Links

Some recent artificial intelligence programs written in LISP have found it convenient to associate *property lists* with list cells as well as symbolic atoms. Since few cells actually have property lists, it is a waste of storage to allocate to every cell a pointer which points to the cell's property list. Therefore, it has been suggested [18] that one bit be set aside in

every cell to indicate whether the cell has a property list. If so, the property list can be found by looking in a hash table, using the *address* of the list cell as the key.

Such a table requires special handling in systems having a relocating garbage collector. Our copying scheme gives each semispace its own hash table, and when a cell is copied over into tospace, its property list pointer is entered in the "to" table under the cell's new address. Then when the copied cell is encountered by the "scan" pointer, its property list pointer is updated along with its normal components. A "CDR-coding" system with two "scan" pointers should also keep a third for tracing property list pointers to prevent property lists from destroying chains of "next"-type cells.

4.8 Reference Counting

In this section we consider whether reference counting can be used as a method of storage reclamation to process lists in real time; i.e. we try to answer the question, at least for the real-time context, is reference counting worth the effort, and if so, under what conditions?

A classical reference count system [25,97] keeps for each cell a count of the number of pointers which point (refer) to that cell; i.e. its in-degree. This *reference count* (*refcount*) is continually updated as pointers to the cell are created and destroyed, and when it drops to zero, the cell is reclaimed. When reclaimed, the refcounts of any daughter cells it points to are decremented, and are also reclaimed if zero, in a recursive manner.

Reference counting appears to be unsuitable for real-time applications because a potentially unbounded amount of work must be done when a cell is let go. However, if a *free stack* is used to keep track of freed objects instead of a *free list*, the newly freed cell is simply pushed onto the free stack. When a cell is needed, it is popped off the stack, the refcounts of its daughters are decremented, and if zero, the daughters are pushed back onto the stack. Then the cell which was popped is returned. In this way, only a bounded amount of work needs to be done on each allocation.

We now consider the storage requirements of a reference counting (RC) system. In addition to the memory for N cells, we also need room for N refcounts and a stack. Since the refcounts can go as high as N, they require approximately the same space as a pointer. So we have:

Maximum RC Space Required ≤ 1.5N + the size of the "free stack"

The worst case stack depth is N. However, whenever a cell is on the stack, its refcount is zero, so we can thread the stack through the unused refcounts! So we now have:

Maximum RC Space Required ≤ 1.5N

Reference count systems have the drawback that directed cycles of pointers cannot be reclaimed. It has been suggested [57,30] that refcounts be used as the "primary" method of reclamation, using garbage collection (GC) as a fallback method when that fails. Since RC will not have to reclaim everything and since the average refcount is often very small, it has also been suggested that a truncated refcount (a bounded counter which sticks at its highest value if it overflows) be used to save space.

We say that garbage in a combination RC and GC system is *ref-degradable* if and only if it can be reclaimed by refcounts alone. Cells whose truncated refcounts are stuck are therefore *non-ref-degradable*.

What is the effect of a dual system in terms of performance? Whatever the RC system is able to recycle puts off flipping that much longer. By the time a flip happens in such a two level system, there is no ref-degradable garbage left in tospace. Therefore, the *turnover* of the semispaces is slowed.

How much memory does the dual system require? If truncated refcounts are used, the free stack cannot be threaded through a cell's refcount because it is not big enough to hold a pointer. Therefore, using this method and assuming only a few bits worth of truncated refcount per cell, we have:

Maximum SRT+RC Space Required ≤ N(2+2m) + RC free stack ≤ N(2.5+2m)

So it appears that we have lost something by adding refcounts (even tiny ones), because we still need room for the free stack.

Let us now examine more closely the average timing of CONS under a pure RC versus a pure SRT system. The average time for CONS under the RC system is the same as the maximum time since there is no freedom in the algorithm. The time for CONS in SRT is c_1k+c_2, where c_1 and c_2 are constants. Now c_2 is simply the time to allocate space from a contiguous block of free storage. Certainly incrementing a pointer is much less complex than popping a cell from a stack, following its pointers, decrementing their refcounts, and if zero, pushing them onto the stack. Therefore, we can choose k small enough[8] so that the average time to perform CONS with our SRT method is smaller than the average time to perform CONS in an RC system.[9] This analysis does not even count the additional time needed to keep the refcounts updated. Of course, the storage required for our "pure" SRT system may be many times the storage of the RC system, but SRT will have a smaller average CONS time.

Since this seems counterintuitive, or at least reactionary (given the current penchant for recycling), we give a rationale for why it is so. Reference counting traces the garbage cells, while normal garbage collection traces the accessible cells. Once the number of garbage cells exceeds the number of accessible cells in an region of storage, it is faster to copy the accessible cells out of the region and recycle it whole. When $m>1$, reference counting cannot compete timewise with garbage collection because RC must trace a cell for every cell allocated while GC traces on the average only a fraction $(1/m)$ of a cell for every cell allocated.

On the other hand, if we wish to minimize storage by making $m<1$, a dual scheme with truncated refcounts should reduce the average CONS time over that in the pure scheme. However, CDR-coded lists and other variable sized objects cannot be easily managed with

---

8. Section 4.3 deals with non-integral k's.

9. We can discount the additional time occasionally required by CAR and CDR in our method because any relocation and pointer updating done by them is work that we have already charged to CONS, and does not have to be repeated.

reference counting because the object at the top of the free stack is not necessarily the right size for the current allocation. Thus, CDR-coding can reduce the storage requirement of a "pure" scheme below that of a "dual" system with the same m. But even on a system with objects of uniform size, we are skeptical whether the increased average efficiency of CONS in the "dual" system will offset the increase in k needed to keep the storage requirements the same as the "pure" system. We conclude that, at least on a real memory computer, *reference counting is probably not a good storage management technique unless one a) has uniformly sized objects; b) uses full counts; and c) guarantees no cycles.*

This is not to say that reference counts are not useful. If the LISP language were extended with a function to return the current refcount of an object, and suitably clean semantics were associated with this function, then one might be able to make use of this information *within the user program* to speed up certain algorithms, such as structure tracing or backtracking, a la Bobrow and Wegbreit [17]. This author is not aware of any language which makes this information available; if it were available, good programmers would certainly find a use for it.

4.9 The Costs of Real-Time List Processing

The amount of storage and time used by a real-time list processing system can be compared with that used by a classical list processing system using garbage collection on tasks not requiring bounded response times. The storage required by a classical non-compacting garbage collector is $N(1+\mu)$, if the system uses the Deutsch-Schorr-Waite (DSW) [57, p. 417-418] marking algorithm, and $N(1.5+\mu)$ if it uses a normal stack, for some positive μ. If CDR-coding is used, copying must be done; the storage requirement is then $Ne(2+2\mu)$, where e is the efficiency of the coding. Since e is near .5 [22], the requirement is about $N(1+\mu)$, so that CDR-coding requires approximately the same space as DSW. Comparing these expressions with those derived earlier for our real-time algorithms, we find that *processing LISP lists in real-time requires no more space than a non-real-time system using DSW*. If larger non-uniformly-sized objects like arrays must be managed,

real-time capability requires no more space than the MFYCA system, since a copying collector is already assumed.

The *average* time requirement for CONS in our real-time system is virtually identical to that in a classical MFYCA system using the same cell representation and the same amount of storage. This is because 1) a classical system can do μN CONS'es after doing a garbage collection which marks N nodes--thus giving an average CONS/mark ratio of μ and allowing us to identify μ with m--and 2) garbage collection in our real-time system is almost identical to that in the MFYCA system, except that it is done incrementally during calls to CONS. In other words, the user program pays for the cost of a cell's reclamation at the time the cell is created by tracing some other cell.

CAR and CDR are a bit slower, because they must test whether the value to be returned is in fromspace. However, as noted above, any cell movement done inside CAR or CDR should not be charged to CAR or CDR because it is work which the collector would otherwise have to do and therefore has already been accounted for in our analysis of CONS. Therefore, CAR and CDR are only slower by the time required for the semispace test.[10]

Since RPLACA, RPLACD, EQ, and ATOM are unchanged from their classical versions, their timings are also unchanged.

The overhead calculated for our serial system can be compared to that in Wadler's parallel system [93]. According to his calculations, a parallel garbage collector requires significantly more total time than a non-parallel collector. But this contradiction disappears when it is realized that his parallel collector continues tracing even in the absence of any cell creation activity. Since our system keys collector activity to cell creation, the collector effort is about the same as on a non-real-time system.

---

10. In Greenblatt's LISP machine [41], the virtual memory map performs the semispace test as an integral part of address translation. Thus on this machine, a successful semispace test requires nary an additional microinstruction!

4.10 Applications

4.10.1 A computer with a real memory of fixed size

This application covers the classical 7090 LISP [64] as well as a LISP for a microcomputer. We conceive of even 16-bit microcomputers utilizing this algorithm for real-time process control or simulation tasks. Each of the list processing primitives is intended to run with interrupts inhibited, so that all interrupt processing can make use of list storage for its buffers and other needs. Multiple processes may also use these primitives so long as CONS, CAR, and CDR are used by one process at a time; i.e. they are protected by one system-wide lock. Of course, the system must be aware of the registers of every process.

For these real memory applications, we want to put as much of the available storage under the management of the algorithm as possible. Thus, both atoms (here we mean the whole LISP atom-complex, not just the print-name) and list nodes are stored in the semispaces. CDR-coding is usually a good idea to save memory, but unless the bit-testing is done in microcode, it may be faster to use normal cells and increase the parameter k to keep the storage size small.

The average CONS time is reduced by putting off flipping until all of the free space in tospace is exhausted, i.e. B=T. Thus, after all moving and tracing is done, i.e. S=B, allocation is trivial until B=T. As a result, the average CONS time in our real-time system is approximately the same as that in a classical system. Of course, with a memory size of 2M, the maximum number of cells that can be safely managed is still $Mk/(k+1)$.

4.10.2 A virtual memory computer

The current epitome of this application is Multics LISP with an address space of 2^{36} ($\approx 10^{11}$) 36-bit words, room for billions of list cells. The problem here is not in reclaiming cells that are let go, but keeping accessible cells compact so that they occupy as few pages of real memory as possible. The MFYCA algorithm does this admirably and ours does almost

as well.

Our scheme is still real-time on a virtual memory computer, but the bounds on the elementary list operations now have the order of magnitude of secondary storage operations.

There are some problems, however. Unlike MFYCA, wherein both semispaces were used only during garbage collection, our method requires that they both be active (i.e. partially in real memory) at all times. This may increase the average working set size. A careful analysis needs to be made of our algorithm in order to estimate the additional cost of incremental garbage collection. Brief consideration tells us that the active address space varies from a minimum of $N(1+m)$ just before a flip to $N(2+2m)$ just after. Since at a flip the user program registers are updated in numerical order, relatively constant pointers should be placed in the lower numbered registers to keep the trace order of constant list structure similar between flips. If the average size of an object is much larger than the size of a pointer, the working set may also be reduced by storing the forwarding addresses in a separate table instead of in the old objects in fromspace [16].

In a virtual memory environment, the active address space will automatically expand and contract in response to changes in the number of accessible cells if 1) FLIP re-adjusts the size of fromspace to $(1+m)$[cells in tospace] just before interchanging the semispaces; and 2) flipping occurs when tracing finishes rather than when B meets T. This policy, plus a smaller k than a real memory computer would use, should give both a fast CONS and a tolerable working set size. The parameter k can also be dynamically adjusted to optimize either running time (including paging) or cost according to some pricing policy by following an analysis similar to that of Hoare and others [55,19,3].

4.10.3 A database management system

We conceive of a huge database having millions of records, which may contain pointers to other records, being managed by our algorithm. Examples of such databases are a bill of materials database for the Apollo Project, or a complete semantic dictionary and thesaurus of English for a language understanding program. Performing a classical

garbage collection on such a databank would be out of the question, since it might require days or weeks to complete, given current disk technology.

Some of these large database systems currently depend on reference counts for storage reclamation, and so do not allow directed cycles of pointers. Since our method performs general garbage collection, this restriction could be dropped. Moreover, given enough space, our algorithm can take even *less time* than a reference count system. When compared with a classical garbage collection system, our method would not save any *total* time in processing transactions against such a data base, but it would avoid the catastrophic consequences of a garbage collection during a period of heavy demand.

This case is very much like case 1, the real memory computer, because we assume that the database is orders of magnitude too big to fit into primary memory and thus that there is little hope for a speedup from the locality of reference effect. "Read memory" and "store memory" instructions here apply to secondary storage; the constant bounds for the elementary operations are now on the order of milliseconds rather than microseconds. Therefore, almost everything that we say about real memory implementations also applies to large database implementations, except that space is cheaper and time is more dear.

4.10.4 A totally new computer architecture

We conceive of an architecture in which a CPU is connected to a *list memory* instead of a random access memory. Machines of this architecture are similar to "linking automata" described by Knuth [57, p. 462-463] and "storage modification machines" described by Schonhage [77]. At the interface between the CPU and the memory sits a bank of *pointer registers*, which point at particular cells in the list memory. Instead of a bus which communicates both addresses and values, with read and write commands, the memory would have only a data bus and commands like CAR, CDR, CONS, RPLACA, RPLACD, and ATOM, whose arguments and returned values would be in the pointer registers. The CPU would not have access to the bit strings stored in the pointer registers, except those which pointed to atoms (objects outside both fromspace and tospace). This restriction is

necessary to keep the CPU from depending upon memory addresses which might be changed by the management algorithm without the CPU's knowledge.

An advantage of such a system over random access memory is the elimination of the huge address bus that is normally needed between the CPU and the memory, since addresses are not dealt with directly by the CPU. As the number of bits on a chip increases, the number of address lines and supporting logic becomes a critical factor. However, since address lines are not available to communicate with other memory chips, we have not yet been able to find a satisfactory way of scaling this memory up.

Our method of garbage collection can also be used with a random access *write-once* memory by appending an extra word to each cell which holds the forwarding address when that cell is eventually moved. Using such a system, the cells in tospace cannot be updated until they are moved to the new tospace after the *next* flip. In other words, *three* semispaces need to be active at all times. In addition to these changes, RPLACA and RPLACD must actually perform a CONS, just like RPLACD occasionally does in our CDR-coding system. Perhaps the write-once property can eliminate the need for transaction journals and backup tapes.

4.11 Conclusions and Future Work

We have exhibited a method for doing list processing on a serial computer in a real-time environment where the time required by all of the elementary list operations must be bounded by a constant which is independent of the number of list cells in use. This algorithm was made possible through: 1) a new proof of correctness of parallel garbage collection based on the assertion that the user program sees only marked cells; 2) the realization that collection effort must be proportional to new cell creation; and 3) the belief that the complex interaction required by these policies makes parallel collection unwieldy. We have also exhibited extensions of this algorithm to handle a user program stack, "CDR-coding", vectors of contiguous words, and hash linking. Therefore, we consider our system to be an attractive alternative to reference counting for real-time storage management

and have shown that, given enough storage, our method will outperform a reference count system, without requiring the topological restrictions of that system.

Our real-time scheme is strikingly similar to the incremental garbage collector proposed independently by Barbacci for a microcoded LISP machine [8]. However, his non-real-time proposal differs in the key points above. Our system will itself appear in microcoded form in Greenblatt's LISP machine [41,11].

There is still some freedom in our algorithm which has not been explored. The *order* in which the cells are traced is not important for the algorithm's correctness or real-time properties. The average properties of the algorithm when run on a virtual memory machine need to be extensively investigated.

The space required by our algorithm may be excessive for some applications. Perhaps a synthesis of Bishop's *area* concept [16] with our method could reduce the memory requirements of a list processing system while preserving the bounded-time properties of the elementary operations.

A garbage collection algorithm can be viewed as a means for converting a Von Neumann-style random access memory (with "side-effects" [64]) into a list memory (without "side-effects"). Perhaps a list memory can be implemented directly in hardware which uses considerably less energy by taking advantage of the lack of side-effects in list operations [12].

5. Garbage Collecting Activities Incrementally

This chapter[1] presents a method by which active objects like actors may be incrementally garbage-collected. This method solves a problem which arises when multiple activities are started in an actor system, and later it is determined that some of them are no longer useful. Rather than allow them to continue wasting system resources, we would like to identify and stop these activities which are no longer relevant to satisfying the current goals of the system.

The best example of this problem occurs when an actor system is used to evaluate an expression in "future" order, which is different from call-by-name, call-by-value, and call-by-need. In future order evaluation, an object called a "future" is created to serve as the value of each expression that is to be evaluated and a separate activity is dedicated to its evaluation. Future order evaluation allows for more parallelism than even the parallel evaluation of arguments discussed in chapter 2, because an argument to a procedure commences being evaluated before the body of the procedure. This argument evaluation proceeds in parallel with the evaluation of the body of the procedure until the procedure finally requires it. As a result, several levels of a recursive procedure may be evaluated in parallel, and many loops written as recursive procedures will be automatically "unrolled" and the different incarnations of the loop body will be evaluated concurrently! Future order evaluation raises a new problem which did not exist in systems with only call-by-name or call-by-value, namely that some futures which were created in the course of evaluation may become irrelevant, i.e. an activity is started to evaluate a future because its result is needed, but after further collateral computation, the activity is deemed unnecessary. This unnecessary activity should be stopped and garbage collected, so as to return its resources to the system.

---

1. This chapter is based on the paper "The Incremental Garbage Collection of Processes" by myself and Carl Hewitt [6].

The problem of irrelevant activities also appears in multiprocessing problem-solving systems which start several processors working on the same problem but with different methods, and return with the solution which finishes first. This *parallel method* strategy also has the problem that the activities which are investigating the losing methods must be identified, cleanly stopped, and their resources re-assigned to more useful tasks.

The solution we propose is that of incremental garbage collection. If the dependency structure of the evaluation plan is explicitly represented in memory as part of the graph memory (like Lisp's heap), a garbage collection algorithm can discover which activities are performing useful work, and which can be recycled.

Call-by-future is implemented by an "eager beaver" evaluator. When an expression of the language is given to the evaluator by the user, the evaluator evaluates it and all of its subexpressions as soon as possible, and in parallel. The evaluator does this by creating and returning for each subexpression a *future*, which is a promise to deliver the value of that subexpression at some later time, if it has a value. Each future can evaluate its subexpression independently and concurrently with other futures because it is created with its own evaluator *activity*, which is dedicated to evaluating that subexpression. When the value of a future is needed explicitly, e.g. by the primitive function "+", the evaluation of the subexpression may or may not be complete. If it is complete, the future's value is immediately available; if not, the requesting activity is forced to wait until the evaluation of the subexpression is finished.

Futures are created recursively in the evaluation of an expression by our eager evaluator whenever it encounters functional application. A new future is created for each argument, resulting in the parallel (collateral) evaluation of those arguments, while the main activity tackles the job of evaluating the function position and applying it to the tuple of argument futures. We call the main evaluator activity the *parent*, while the futures it directly creates are its *offspring*.

More precisely, a *future* is a triple (activity, cell, waiting room), where *activity* is the activity charged with evaluating an argument expression in its proper environment, *cell* is a cell actor, private to the future, which will save the value of the argument when it is ready,

to avoid recomputing it, and *waiting room* is a set of activity continuations which are waiting for the value of this future.

When the future is created, its activity starts evaluating the subexpression in the given environment. If any other activity needs the value of this future before it is ready, the requesting activity puts its continuation in the waiting room of the future. When the value promised by the future is ready, its activity stores that value into the future's cell, sends wakeup messages to all of the activity continuations in the future's waiting room, and goes away. Henceforth, any activity needing this future's value can find it in the future's cell, without waiting or performing any further computation.

Notice that eager evaluation is different from lazy evaluation [94,91,46,50,36] of the expression in that the latter is designed to delay evaluation of the expression until the value is needed while a future immediately dedicates an activity to evaluating the expression. This difference is both a strength and a weakness of eager evaluation.

The main problem with our eager interpreter is that it can be wasteful, because it anticipates which values are going to be required to compute the final result. For example, an activity may be assigned to the computation of a future whose value will never be needed; in this case, we say that the activity is *irrelevant*. However, since the *a priori* determination of irrelevancy seems undecidable, all activities must proceed until irrelevancy can be proven. If there were no way to determine irrelevancy *a posteriori*, these irrelevant activities could tie up a significant amount of computing power. For example, if an activity were assigned to evaluate a non-terminating expression, its computational power would be lost to the system forever! We argue in the following sections that the "garbage collection" of passive storage can be extended to the reclamation of these irrelevant active activities. Furthermore, we show that this garbage collection can be done *incrementally*, thus eliminating the long delays classically associated with garbage collection.

5.1 Garbage Collecting Irrelevant Futures

A classical garbage collector for passive storage starts by marking the *root* of the heap of passive storage nodes, and proceeds by propagating marks from marked nodes to their offspring, until there are no unmarked nodes with a marked parent. Upon the completion of this process, any nodes which are still unmarked are not accessible from the root; hence they are declared garbage and returned to the list of available free nodes.

The key to garbage collecting *activities* is that an activity's continuation is addressable as a vector of words in the common address space of all the processors, but distinguished with a special type code. This vector stores the acquaintances of the continuation. We claim that activities whose continuations become inaccessible from the root are irrelevant and should be reclaimed. The top-level activity--that assigned to the top level future--is always relevant since the user expects an answer, and therefore it is always directly accessible from the root of the heap. Any offspring of this future whose values are still required are accessible to it. Hence by induction, relevant activities remain accessible from the root. If a future becomes inaccessible from the root, then no other activity can access its value--even when it is finally ready--and hence the future and its activity are irrelevant.

In order that all irrelevant activities be identified as soon as possible, we must ensure that all activities classified as accessible are truly relevant to the computation. An example of an activity which is accessible but irrelevant is that of a *loiterer*, i.e. an activity whose continuation is accessible only through the "waiting room" of some future. A loiterer is waiting for the value of one or more futures, but the loiterer's value is not needed by any other activity. Loiterers cannot be immediately garbage collected because of the outstanding waiting-room pointers to them. However, when the loiterer is eventually restarted and forgotten by the waiting room, it will then become inaccessible from the root of the heap and will be picked up by the next garbage collection. Hence, waiting-room accessibility is a second-class form of accessibility which will not protect a loiterer from eventually being garbage collected.

If busy waiting is used, waiting rooms are not necessary, and thus there will be no loiterers. However, busy waiting requires that a high price be paid for communication channels between the waitor and the waitee, because the incessant queries clog these channels.

Garbage collection is made incremental by using some of the ideas from the previous chapter. The mark phase of our incremental garbage collector process employs three colors for every object--*white*, *grey*, and *black*. Intuitively, *white* nodes are not yet known to be accessible, *grey* nodes are known to be accessible, but whose offspring have not yet been checked, and *black* nodes are accessible, and have accessible offspring. Initially, all nodes (including actors) are white. A white node is made grey by *shading* it; i.e. making it "at least grey" [31], while a grey node is *marked* by shading its offspring and making the node black--both indivisible processes. Marking is initiated by stopping all message transmission and shading the root. Marking proceeds by finding a grey node, shading its offspring, then making that node black. When there are no more grey nodes, garbage collection is done; all still-white nodes are then emancipated and the colors white and black switch interpretations.

Although all activity must be stopped when garbage collection is begun, an activity can be restarted as soon as it has been blackened by the collector. Since the top-level activity is pointed at directly by the root of the heap, it is restarted almost immediately. It should be obvious that when an activity first becomes black, it cannot point directly at a white node. We wish to preserve this assertion. Therefore, whenever a running black activity is about to violate it--by accessing a white acquaintance--it immediately shades the white actor before proceeding. Furthermore, every new actor the activity creates is created black. The intuitive rationale behind these policies is that *so far as any black activity is concerned, the garbage collection has already finished.* Furthermore, *the actors which are found accessible by the garbage collector are exactly those which were accessible at the time the garbage collection was started.*

We prove the correctness of this garbage collector informally. The garbage collector is given a head start on all of the activities because they are stopped when it is started. When an activity is restarted, it is black, and everything it sees is at least grey, hence it is in the

collector's wake. Whenever an activity attempts to catch up to the collector by accessing an acquaintance, that actor is immediately shaded. Therefore, the activity can never pass or even catch the collector. Since the collector has already traced any actor an activity may have as an acquaintance, the activity cannot affect the connectivity of the graph that the collector sees. Because white or grey activities are not allowed to run, any created actors are black, and since actors darken monotonically, the number of white actors must monotonically decrease, proving termination.

Our garbage collector has only one phase--the mark phase--because it uses the Cheney algorithm which marks and copies in one operation. This algorithm copies accessible list structures from an "old semispace" into a "new semispace". As each node is copied, a "forwarding address" is left at its old address in the old semispace. A "scan" pointer linearly scans the new semispace, while updating the pointers of newly moved nodes by moving the nodes they point to. The correspondence between our coloring scheme and this algorithm is this: *white* actors are those which reside in the old semispace; *grey* actors are those which have been copied to the new semispace, but whose acquaintances have not been moved to the new semispace (i.e. have not yet been encountered by the scan pointer in the Cheney algorithm); and *black* actors are those which have been both moved and updated (i.e. are behind the scan pointer). When scanning is done (i.e. there are no more grey actors and all accessible actors have been copied), the old and new semispaces then interchange roles. Reallocating processors is simple; all processors are withdrawn at the start of garbage collection, and are allocated to each activity as it is blackened. Thus, when the garbage collection has finished, all and only relevant (=accessible) activities have been restarted.

The restriction that white or grey activities cannot run can be relaxed to allow white activities to run *so long as a white activity does not cause a black actor to point to a white one.* This can only happen if the white activity is trying to perform a side-effect (e.g. a "store!" operation) on a black actor. If operations of this type are suspended until either the activity either becomes black or is garbage-collected, then proper garbage collector operation can be ensured, and convergence guaranteed. Under these conditions, a activity creates new actors of its own color, i.e. white activities create only white actors. When a white activity is

encountered by the garbage collector, it must stop and allow itself to be colored black before continuing.

The notion that actors must be marked as well as storage may explain some of the trouble that Dijkstra and Lamport had when trying to prove their parallel garbage collection algorithm correct [31,32,58,59]. Since their algorithm does not mark a user process by coloring it black (thereby prohibiting it from directly touching white nodes), and allows these white processes to run, the proof that the algorithm collects only and all garbage is long and very subtle (see [59]).

5.2 Coroutines and Generators

One problem with our "eager beaver" evaluator is that some expressions which have no finite values will continue to be evaluated without mercy. Consider, for example, the infinite sequence of squares of integers 0,1,4,9,... We give in Figure 19 a set of LISP-like functions for computing such a list.

The evaluation of "(squares-beginning-with 0)" will start off a future evaluating "(cons ...)", which will start up another future evaluating "(squares-beginning-with 1)", and so forth. Since this computation will not terminate, we might worry whether anything useful

---

Fig. 19. An Infinite Sequence of Squares

```
squares-begining-with ≡
  (λx. (cons (* x x) (squares-begining-with (+ x 1))))  ; Compute an element.

cons ≡ (λx y.                                           ; Define CONS function.
          (λmsg.
             (if (= msg 'car) x
                 (= msg 'cdr) y)))

car ≡ (λx. (x 'car))                                    ; Ask for first component.

cdr ≡ (λx. (x 'cdr))                                    ; Ask for second component.

list-of-squares ≡ (squares-beginning-with 0)            ; Start the recursion.
```

will ever get done. One way to ensure that this computation will not clog the system is to convert it into a "lazy" computation [94,91,46,36] by only allowing it to proceed past a point in the infinite list when someone forces it to go that far. This can be easily done by performing a lambda abstraction on the expression whose evaluation is to be delayed. (See Figure 20). Since our evaluator will not try to further evaluate a λ-expression, this will protect its body from evaluation by our eager beavers.

However, this technique is not really necessary if we use an *exponential scheduler* for the proportion of effort assigned to each activity. This scheduler operates recursively by assigning 100% of the system effort to the top-level future, and whenever this future spawns new futures, it allocates only 50% of its allowed effort to its offspring. While an activity is in the waiting room of a future, it lends it processing effort to the computation of that future. However, a future which finishes returns its effort to helping the system--not its siblings. Now the set of futures can be ordered according to who created whom and this ordering forms a tree. As a result of our exponential scheduling, the further down in this tree a future is from the top-level future, the lower its share of the computational resources. Therefore, as our eager beavers produce more squares, they become exponentially more discouraged. But if other activities enter the waiting room for the square of a large number, they lend their encouragement to its computation.

Fig. 20. A Lazy Sequence of Squares

```
squares-begining-with' ≡
  (λx. (cons (* x x)
             (λmsg. ((squares-begining-with' (+ x 1)) msg))))  ; Protect from early evaluation.
```

Call-by-future evaluation provides for the maximal concurrency possible in evaluating the expressions of a language. It can provide more parallelism than current data flow machines [28,4] or "eventual values" [53]. For example consider the following program which computes the square root of the sum of the squares of its arguments:

f ≡ (λx y. (square-root (+ (* x x) (* y y))))

Note that in computing the value of an expression such as the following (k 3 (f (h 3) (g 4))) that the square of (h 3) can be performed in parallel with the square of (g 4). In addition the square root of the sum of these values might be performed after the function k has been entered! Thus there is a great deal of potential concurrency in the evaluation of the above expression.

In an evaluator which uses call-by-future for CONS, the obvious program for MAPCAR (the LISP analog of APL's parallel application of a function to a vector of arguments) will automatically do all of the function applications in parallel in a "pipe-lined" fashion. However, with an exponential scheduler the values earlier in the list will be accorded more effort than the later ones.

This scheduler is not omniscient, though, and system effort will still have to be reallocated by the garbage collector as it discovers irrelevant activities and returns their computing power to help with still relevant tasks.

5.3 Time and Space

"Lazy" evaluation [94,46,36] is an optimal strategy [91,13] for evaluating λ-calculus expressions on a single processor, in the sense that the minimum number of reductions (procedure calls) are made. However, when more than one processor is available to evaluate the expression, it is not clear what strategy would be optimal. If nothing is known about the particular expression being evaluated, we conjecture that any reasonable strategy must allocate one processor to lazy evaluation, with the other processors performing eager

evaluation. We believe that our "eager beaver" evaluator implements this policy, and unless the processors interfere with one another excessively, a computation must always run faster with an eager evaluator running on multiple processors than a lazy evaluator running on a single processor. If there are not enough processors to allocate one for every future, then we believe that our "exponential scheduling" policy will do a good job of dynamically allocating processor effort where it is most needed.

Although the universal creation of futures should reduce the time necessary to evaluate an expression, we must consider how the space requirements of this method compare with other methods. The space requirements of futures are hard to calculate because under certain schedules, future order evaluation approximates call-by-value, while with other schedules, it is equivalent to call-by-need (the same as call-by-value, but an argument is evaluated only once). In the worst case, the space requirements of futures can be arbitrarily bad, depending upon the relative speed of the processors assigned to non-terminating futures.

5.4 The Power of Futures

The intuitive semantics associated with a future is that it runs asynchronously with its parent's evaluation. This effect can be achieved by either assigning a different processor to each future, or by multiplexing all of the futures on a few processors. Given one such implementation, the language can easily be extended [65] with a construct having the following form: "(EITHER $<e_1>$ $<e_2>$ ... $<e_n>$)" means evaluate the expressions $<e_i>$ in parallel and return the value of "the first one that finishes". Ward [95] shows how to give a Scott-type lattice semantics for a generalization of this construct. He starts with a power-set of the base domain and gives it the usual subset lattice structure, then extends each primitive function to operate on *sets* of elements from the base domain in the obvious way, and finally defines the result of his construct to be the *least upper bound* (*LUB*) of all the $<e_i>$ in the subset lattice. Our EITHER construct is approximated[2] by spawning futures for all the $<e_i>$, and polling them with the parent activities until the first one finishes. At

that point, its answer is returned as the value of the "EITHER" expression, and the other futures become inaccessible from the root of the heap.

In Figure 21 we give several examples of the power of the "EITHER" construct:

The first example is that of a numeric product routine whose value is zero if either of its arguments are zero, even if the non-zero argument is undefined. The second example is an integration routine for use in a symbolic manipulation language like Macsyma, where there is a relatively fast heuristic integration routine which looks for common special cases, and a general but slow decision procedure called the Risch algorithm. Since the values of both methods are guaranteed to be the same (assuming that they perform integration properly), we need not worry about the possibility of non-determinacy of the value of this expression (i.e. non-singleton subsets of the base domain in Ward's lattice model).

One may ask what the power of such an "EITHER" construct is; i.e. does it increase the expressive power of the language in which it is embedded? A partial answer to this question has been given with respect to "uninterpreted" schemata. Uninterpreted schemata answer questions about the expressive power of programming language constructs which are implicit in the language, rather than being simulated. For example, one can compare the power of recursion versus iteration in a context where stacks cannot be simulated. Hewitt and Paterson [48] have shown that uninterpreted "parallel" schemata are strictly more powerful than recursive schemata. The essence of this difference is that parallel

---

Fig. 21. Examples of the EITHER Construct

```
(multiply x y) ≡ (EITHER (if x=0 then 0 else (loop))
                         (if y=0 then 0 else (loop))
                         (* x y))

(integrate exp bound-variable) ≡
  (EITHER (fast-heuristic-integrate exp bound-variable)
          (Risch-integrate exp bound-variable))
```

---

2. This implementation is only an approximation because only singleton sets of elements of the base domain can ever be returned.

schemata can simulate non-deterministic computation without becoming side-tracked in some infinite branch of the computation. This simulation is possible because the parallel schema can follow all of the non-deterministic branches in parallel.

Also, Ward [95] has shown that the "EITHER" construct strictly increases the power of the λ-calculus [21,26] in the sense that there exist functions over the base domain which are inexpressible without "EITHER", but are trivially expressible with it.

5.5 Shared Data Bases

The advantage that garbage collection has over the explicit killing of activities becomes apparent when parallel activities have access to a shared data base. These data bases are usually protected from inconsistency (due to simultaneous update) by a mutual exclusion method. However, if some activity were to be killed while it was inside such a data base, the data base would remain locked, and hence unresponsive to the other activities requesting access.

The solution we propose is for the data base to always keep a list of pointers to the activities which it has currently inside. In this way, an otherwise irrelevant activity will be accessible so long as it is inside an accessible data base. However, the moment it emerges, it will be forgotten by the data base, and subject to reclamation by garbage collection. The *crowds* component of a *serializer*, a synchronization construct designed to manage parallel access to a shared data base [51], automatically performs such bookkeeping.

5.6 Conclusions

We have presented a method for managing the allocation of processors as well as storage to the subcomputations of a computation in a way that tries to minimize the elapsed time required. This is done by anticipating which subcomputations will be needed and starting them running in parallel, before the results they compute are needed. Because of this anticipation, subcomputations may be started whose results are not needed, and our incremental garbage collection method identifies and revokes these allocations of storage

and processing power.

Some of the early thinking about call-by-future was done several years ago by J. Rumbaugh who was one of the first to realize that futures offer a maximum of concurrency in the execution of a program without introducing the usual pitfalls of timing errors, starvation, and deadlock. Unfortunately he did not have time to include this material in his thesis [75]. Peter Hibbard [53] has independently discovered these virtues of futures. The main original contributions of this chapter are our proposal for an exponential scheduler for "eager" evaluation and the methodology for using incremental garbage collection to reclaim irrelevant activities and redirect the scheduling priorities of activities working to produce the values of futures. Concepts similar to that of "futures" have been independently proposed by Friedman and Wise [37] and implemented by Hibbard [53].[3] Henry Lieberman, working on implementations of actor language PLASMA [50], has actually implemented several of the suggestions of this chapter.

The scheme presented here assumes that all of the processors reside in a common, global address space, like that of HYDRA [98]. Since networks of local address spaces look promising for the future, methods for garbage collecting those systems need to be developed.

---

3. However, since Algol-68 does not support "returned functional values", A scheme in the language need not use garbage collection to discover irrelevant "eventuals". They can be coerced into values before being returned as the value of a procedure, and hence processor allocation can use a LIFO scheme like that used for storage of the activation records on the stack. However a certain amount of concurrency can be lost by enforcing this coercion.

6. Conclusions and Further Research

This thesis has been concerned with a precise specification of the Actor theory and with possible mechanisms for mapping a system having Actors as primitive objects onto current hardware.

The Actor theory does not try to accurately describe every detail of a distributed system, because in creating a model for some phenomenon, one must choose which of its features to emphasize and which to suppress. The actor model ignores the *sending* of messages and concentrates instead on their *receipt*, because the receipt of messages is more interesting due to the non-determinacy involved. Our model also ignores unreliabilities in the network by assuming that every message is eventually received, because we believe that the issue of transmission errors is separable from our current concerns. The actor model of computation as a simple partial order ignores the issue of "real time", i.e. time which can be measured, and concentrates only on the orderings of events. However, in this thesis we are implicitly assuming that the receipt of a message by an actor requires only a bounded amount of computation. Hence, each event should require only a brief instant of real time to complete. Finally, issues of representation have been neglected in favor of issues of behavior; e.g. a particular concrete representation for actors and messages has not been presented.

However, no matter what representations are chosen for the implementation of an actor system, certain problems in the management of system resources will arise. In chapters 3 and 4 we argue that such an implementation will require a garbage-collected heap for storage management and that for real-time performance an incremental method of garbage collection will be necessary. Such a method is presented together with an exhaustive comparison of it with other alternatives.

In chapter 5, we argue that in a distributed parallel processing system the resources of the CPU's can be squandered exploring avenues of computation that have ceased to be of relevance to satisfying the main goals of the system. We advocate a garbage collection approach to this problem in which the garbage collector discovers and stops these irrelevant

lines of computation. As the number of CPU's in parallel systems continues to rise, programmers will want to use more of this computational power for such speculative computation, and the need for our garbage collection methods will grow.

Looking into the future, a major problem to be overcome in the management of the resources of a distributed system is the garbage collection of objects in separate address spaces. Before this problem can be solved, however, the issues of object *naming* in a distributed system must first be resolved. In systems where names have global significance, Bishop's *area* concept [16] may be appropriate, in which case his algorithms for garbage collection are of interest. Reed's proposal for *object-families* [72] may also provide a proper model for naming in distributed systems.

Now that we have provided a firm foundation for the implementation of real-time actor systems by showing how the basic events (message receipts) can be performed in a bounded amount of time, research into good scheduling strategies for such systems should follow. Also, techniques will be needed for deriving and proving time bounds for complex activities requiring many hundreds of events.

References

1. Aho, A. V., Hopcroft, J. E., and Ullman, J. D. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, Reading, Mass., 1974.

2. Arnborg, S. "Storage Administration in a Virtual Memory SIMULA System". *BIT* 12 (1972), 125-141.

3. Arnborg, S. "Optimal Memory Management in a System with Garbage Collection". *BIT* 14 (1974), 375-381.

4. Arvind and Gostelow, K. "Some Relationships between Asynchronous Interpreters of a Dataflow Language" IFIP Working Conference on Formal Description of Programming Concepts. 31 July to 5 August, 1977.

5. Baker, H. G., Jr. "List Processing in Real Time on a Serial Computer". AI Working Paper 139, MIT AI Lab., Feb. 1977, also to appear in *CACM*.

6. Baker, H. G. Jr., and Hewitt, C. "The Incremental Garbage Collection of Processes". *ACM SIGART-SIGPLAN Symposium*, Roch., N.Y., Aug. 1977.

7. Banks, E. R. "Information Processing and Transmission in Cellular Automata". TR-81, MIT Project MAC, Jan. 1971.

8. Barbacci, M. "A LISP Processor for C.ai". Memo CMU-CS-71-103, Computer Science Dept., Carnegie-Mellon University, 1971.

9. Barnes, G. H., et al. "The ILLIAC IV Computer". *IEEE Transactions*, C17,8 (Aug. 1968).

10. Batcher, K. E. "Sorting Networks and their Applications". 1968 SJCC, April 1968, 307-314.

11. Bawden, A., Greenblatt, R., Holloway, J., Knight, T., Moon, D., Weinreb, D. "LISP Machine Progress Report". Memo 444, MIT AI Lab., Aug. 1977.

12. Bennett, C. H. "Logical Reversibility of Computation". *IBM J. Res. Develop.* 17 (1973), 525.

13. Berry, Gerard and Levy, Jean-Jacques. "Minimal and Optimal Computations of Recursive Programs". Record of 1977 Conference on Principles of Programming Languages, Jan. 1977, 215-226.

14. Berzins, Valdis. "An Independence Result for Actor Laws". Computation Structures Note 34, MIT Lab. for Comp. Sci., Dec. 1977.

15. Birtwistle, G. M., Dahl, O.-J., Myhrhaug, B, and Nygaard, K. *Simula Begin*. Auerbach, Phil., Pa., 1973.

16. Bishop, P. B. "Computer Systems with a Very Large Address Space and Garbage Collection". Ph.D. Thesis, M.I.T. Department of Electrical Engineering and Computer Science. Also TR-178, MIT LCS, May 1977.

17. Bobrow, D. G. and Wegbreit, B. "A Model and Stack Implementation of Multiple Environments". *CACM* 16,10 (Oct. 1973), 591-603.

18. Bobrow, D. G. "A Note on Hash Linking". *CACM* 18,7 (July 1975),413-415.

19. Campbell, J. A. "Optimal Use of Storage in a Simple Model of Garbage Collection". *Info. Proc. Letters* 3, No. 2, Nov., 1974, 37-38.

20. Cheney, C. J. "A Nonrecursive List Compacting Algorithm". *CACM* 13,11 (Nov. 1970), 677-678.

21. Church, A. "The Calculi of Lambda Conversion". *Annals of Mathematics Studies*, Princeton University Press, 1941.

22. Clark, D. W. and Green, C. C. "An Empirical Study of List Structure in Lisp". *CACM* 20,2 (Feb. 1977), 78-87.

23. Clinger, W. Unpublished 6.835 class notes, MIT EECS Dept., Dec. 1977.

24. Coffman, E. G. Jr. (ed.) *Computer and Job-Shop Scheduling Theory*. Wiley and Sons, New York, 1976.

25. Collins, G. E. "A Method for Overlapping and Erasure of Lists". *CACM* 3,12 (Dec. 1960), 655-657.

26. Curry, H. B., and Feys, R. *Combinatory Logic*, Amsterdam, 1958.

27. Dahl, O.-J. and Nygaard, K. "SIMULA--an ALGOL-Based Simulation Language". *CACM* 9,9 (Sept. 1966), 671-678.

28. Dennis, J. and Misunas, D. P. "A Preliminary Architecture for a Basic Data-Flow Processor. In 2nd IEEE Symposium on Computer Architecture., N.Y. Jan. 1975, 126-132.

29. Deutsch, L. P. "A LISP Machine with Very Compact Programs". IJCAI 3, Stanford, Ca., Aug. 1973.

30. Deutsch, L. P. and Bobrow, D. G. "An Efficient, Incremental, Automatic Garbage Collector". *CACM* 19,9 (Sept. 1976), 522-526.

31. Dijkstra, E. W., Lamport, L., Martin, A. J., Scholten, C. S., Steffens, E. F. M. "On-the-fly Garbage Collection: An Exercise in Cooperation". E. W. Dijkstra note EWD496, June 1975.

32. Dijkstra, E. W. "After Many a Sobering Experience". E. W. Dijkstra note EWD500.

33. Erman, L. D. and Lesser, V. R. "A Multi-level Organization for Problem Solving using Many, Diverse, Cooperating Sources of Knowledge". IJCAI-75, Sept. 1975, 483-490.

34. Fenichel, R. R., and Yochelson, J. C. "A LISP Garbage-Collector for Virtual-Memory Computer Systems". *CACM* 12,11 (Nov. 1969), 611-612.

35. Fischer, M. J. "Lambda Calculus Schemata". Proceedings of ACM Conference on Proving Assertions about Programs. *SIGPLAN Notices* Jan. 1972.

36. Friedman, D. P. and Wise, D. S. "CONS should not evaluate its arguments". In S. Michaelson and R. Milner (eds.), *Automata, Languages and Programming*, Edinburgh University Press, Edinburgh (1976), 257-284.

37. Friedman, D. P. and Wise, D. S. "The Impact of Applicative Programming on Multiprocessing". 1976 Int. Conf. on Parallel Processing, 263-272. Also *IEEE Trans. on Comps.*, to appear.

38. Gardner, M. (ed.) "Mathematical Games". *Scientific American*, New York, Oct. 1970.

39. Goldberg, A. and Kay, A. (eds.) *SMALLTALK-72 Instruction Manual*. SSL 76-6, Xerox PARC, Palo Alto, Ca., March 1976.

40. Goto, E. and Kanada, Y. "Hashing lemmas on time complexity with applications to formula manipulation." SYMSAC 76 (ACM), New York.

41. Greenblatt, R. "The LISP Machine". AI Working Paper 79, M.I.T. A.I. Lab., Nov. 1974.

42. Greenblatt, R. Private communication, Feb. 1977.

43. Greif, I. "Semantics of Communicating Parallel Processes, PhD Thesis, TR-154, MIT Project MAC, Sept. 1975.

44. Greif, I. and Hewitt, C. "Actor Semantics of PLANNER-73". *ACM SIGPLAN-SIGACT Conf.*, Palo Alto, Ca., Jan. 1975.

45. Halstead, Robert H. Jr. "Multiprocessor Implementations of Message-Passing Systems". S.M. Thesis, MIT, Feb. 1978.

46. Henderson, P. and Morris J. H. "A Lazy Evaluator" In Proceedings of 3rd ACM Symposium on Principles of Programming Languages. (1976), 95-103.

47. Hennie, F. C. *Iterative Arrays of Logical Circuits.* MIT Press, Camb., Ma., 1961.

48. Hewitt, C. and Paterson, M. "Comparative Schematology". Record of Project MAC Conference on Concurrent Systems and Parallel Computation, June 1970.

49. Hewitt, Carl, et al. "Behavioral Semantics of Non-recursive Control Structures". Proc. *Colloque sur la Programmation.* Lecture Notes in Computer Science No. 19. Springer-Verlag, 1974.

50. Hewitt, C. "Viewing Control Structures as Patterns of Passing Messages" WP 92, MIT AI Lab., Dec. 1975. Accepted for publication in the A.I. Journal.

51. Hewitt, C. and Atkinson, R. ,"Parallelism and Synchronization in Actor Systems". Record of 1977 Conference on Principles of Programming Languages, Jan. 17-19, 1977, L.A., Cal., 267-280.

52. Hewitt, C. and Baker, H. "Actors and Continuous Functionals". Memo 436A, MIT AI Lab., July 1977.

53. Hibbard, P. "Parallel Processing Facilities". in *New Directions in Algorithmic Languages*, (ed.) Stephen A. Schuman, IRIA, 1976, 1-7.

54. Hoare, C. A. R. "Monitors: An Operating System Structuring Concept". Stanford University, 1973.

55. Hoare, C. A. R. "Optimization of Store Size for Garbage Collection". *Info. Proc. Letters* 2 (1974), 165-166.

57. Holt, A., et al. *Final Report of the Information System Theory Project*, TR-68-305, RADC, Griffis AFB, N.Y., Sept. 1968.

57. Knuth, D. E. *The Art of Computer Programming, Vol. 1, Fundamental Algorithms.* Addison-Wesley, Reading, Mass. 1968.

58. Lamport, L. "Garbage Collection with Multiple Processes: An Exercise in Parallelism". Mass. Computer Associates, CA-7602-2511, Feb. 1976.

59. Lamport, L. "On-the-fly Garbage Collection: Once More with Rigor". Mass. Computer Associates, CA-7508-1611, Aug. 1975.

60. Landin, Peter J. "A Correspondence between ALGOL 60 and Church's Lambda-Notation". *CACM* 8,2-3 (Feb. and March 1965).

61. Liskov, B., Snyder, A., Atkinson, R., and Schaffert, C. "Abstraction Mechanisms in CLU". *CACM* 20,8 (Aug. 1977), 564-576.

62. Liskov, B. Private communication, Feb. 1978.

63. Liu, C. L. and Layland, J. W. "Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment". *JACM* 20,1 (1973), 46-61.

64. McCarthy, John, et al. *LISP 1.5 Programmer's Manual.* MIT Press, Cambridge, Mass., 1965.

65. McCarthy, J. "A Basis of the Mathematical Theory of Computation" In P. Braffort and D. Hirschberg (eds.) Programming Systems and Languages. McGraw-Hill, New York(1967), 455-480.

66. Minsky, M. L. "A LISP Garbage Collector Algorithm Using Serial Secondary Storage". A.I. Memo 58, M.I.T. A.I. Lab., Oct. 1963.

67. Moon, David A. *MACLISP Reference Manual.* Project MAC, MIT Cambridge, Mass., December 1975.

68. Muller, K. G. "On the Feasibility of Concurrent Garbage Collection". Ph.D. Thesis, Technische Hogeschool Delft, The Netherlands, March 1976, (In English).

69. Newell, A., et al. *Information Processing Language V Manual.* Second edition, Prentice-Hall, Englewood Cliffs, N.J., 1964.

70. Plotkin, G. D. "A Powerdomain Construction". *SIAM J. Comput.* 5,3 (Sept. 1976), 452-487.

71. Pratt, V. R. "The Competence/Performance Dichotomy in Programming". Memo 400, MIT AI Lab., Jan. 1977.

72. Reed, D. P. "Naming of Objects in a Distributed Autonomous Computer System". PhD thesis proposal, MIT EECS Dept., July 1977.

73. Reynolds, John C. "Definitional Interpreters for Higher Order Programming Languages". ACM Conference Proceedings 1972.

74. Rivest, R. and Pratt, V. "The Mutual Exclusion Problem for Unreliable Processes". *17th IEEE Symp. on the Founds. of Comp. Sci.*, Oct. 1976, 1-8.

75. Rumbaugh, J. E. "A Parallel Asynchronous Computer Architecture for Data Flow Programs" Ph.D. dissertation, M.I.T. May 1975. MAC TR-150.

76. Sassa, M. and Goto, E. "A hashing method for fast set operations." *Information Processing Letters* 5, 1976, 31-34.

77. Schonhage, A. "Real-Time Simulation of Multidimensional Turing Machines by Storage Modification Machines". MAC TM-37, Project MAC, M.I.T., Dec. 1973.

78. Scott, D. "Outline of a Mathematical Theory of Computation". *4th Princeton Conf. on Inf. Sci. and Sys.*, 1970, 169-176.

79. Scott, D. "Data Types as Lattices". *SIAM J. Comput.* 5,3 (Sept. 1976), 522-587.

80. Steele, G. L. Jr. "Multiprocessing Compactifying Garbage Collection". *CACM* 18, 9 (Sept. 1975), 495-508.

81. Steele, G. L. Jr., and Sussman, G. J. "LAMBDA: The Ultimate Imperative". Memo 353, MIT AI Lab., March, 1976.

82. Steele, G. L. Jr. "LAMBDA: The Ultimate Declarative". Memo 379, MIT AI Lab., Nov. 1976.

83. Steele, G. L. Jr. Private communication, March 1977.

84. Steele, G. L. Jr. "Debunking the 'Expensive Procedure Call' Myth or, Procedure Call Implementations Considered Harmful or, Lambda: the Ultimate GOTO". Memo 443, MIT AI Lab., Oct. 1977.

85. Steiger, R. J. "Actor Machine Architecture". S.M. Thesis, E.E. Dept., MIT, May 1974.

86. Strachey, C., and Wadsworth, C. P. "Continuations: A Mathematical Semantics for Handling Full Jumps". Tech. Monograph PRG-11, Oxford U. Computing Lab., Jan. 1974.

87. Sullivan, H. and Bashkow T. R. "A Large Scale, Homogeneous, Fully Distributed Parallel Machine". Proc. of Fourth Annual Symposium on Computer Architecture. March 1977, 105-117.

88. Teitelman, W. et. al. *INTERLISP Reference Manual.* Xerox PARC, Palo Alto, Cal., 1974.

89. Tessler, G. and Enea H. J. "A Language Design for Concurrent Processes". In Proc. 1968 SJCC.

90. Ullman, J. D. "Polynomial Complete Scheduling Problems". Proc. 4th Symp. on Oper. Sys. Princ., Oct. 1973.

91. Vuillemin, Jean. "Correct and Optimal Implementations of Recursion in a Simple Programming Language". *JCSS* 9 (1974), 332-354.

92. Vuillemin, Jean. "A Data Structure for Manipulating Priority Queues". TR 182, Dept. d'Informatique, U. de Paris-Sud 19405-ORSAY, France, March 1976.

93. Wadler, P. L. "Analysis of an Algorithm for Real-Time Garbage Collection". *CACM* 19, 9 (Sept. 1976), 491-500.

94. Wadsworth, C. "Semantics and Pragmatics of the Lambda-Calculus" Ph.D. dissertation, Oxford(1971).

95. Ward, S. A. "Functional Domains of Applicative Languages". MAC TR-136, Project MAC, MIT, Sept. 1974.

96. Ward, S. A. and Halstead, R. "A Syntactic Theory of Message-Passing". in progress, MIT LCS.

97. Weizenbaum, J. "Symmetric List Processor". *CACM* 6,9 (Sept. 1963), 524-544.

98. Wulf, W., et al. "HYDRA: The Kernel of a Multiprocessor Operating System". *CACM* 17,6 (June 1974), 337-345.

Biographical Note

Henry Givens Baker, Jr. was born in Hutchinson, Kansas on June 8, 1947. He graduated in 1965 from Walnut Hills H.S. in Cincinnati, Ohio, and got his baccalaureate in Electrical Engineering from the Massachusetts Institute of Technology in June, 1969.

Mr. Baker then received a commission in the Public Health Service and was stationed at the Northeaster Radiological Health Laboratory in Winchester, Massachusetts. In February of 1970, he assumed a full time position as operations research consultant and systems designer for the Palm Beach Company of Cincinnati, Ohio, for which he had done prior consulting.

In September of 1971, Mr. Baker began graduate work in Computer Science at M.I.T. and received his S.M. and E.E. degrees in June, 1973 with a thesis entitled "Equivalence Problems of Petri Nets". After a one year hiatus as Instructor of Computer Science and Engineering at the University of Pennsylvania, he resumed his graduate studies at M.I.T. and received his Ph.D. degree in June, 1978, with a minor in pure mathematics.

Dr. Baker is a member of Tau Beta Pi, Eta Kappa Nu, Sigma Xi, and the Association of Computing Machinery. His interests include running, folk dancing, bicycling, personal computers and chess.

Dr. Baker will now join the faculty of the University of Rochester as Assistant Professor in Computer Science.

Official Distribution List

Defense Documentation Center
Cameron Station
Alexandria, Virginia 22314 — 12 copies

Office of Naval Research
Information Systems Program
Code 437
Arlington, Virginia 22217 — 2 copies

Office of Naval Research
Code 102IP
Arlington, Virginia 22217 — 6 copies

Office of Naval Research
Branch Office/Boston
495 Summer Street
Boston, Massachusetts 02210 — 1 copy

Office of Naval Research
Branch Office/Chicago
536 South Clark Street
Chicago, Illinois 60605 — 1 copy

Office of Naval Research
Branch Office/Pasadena
1030 East Green Street
Pasadena, California 91106 — 1 copy

New York Area Office
715 Broadway - 5th floor
New York, N. Y. 10003 — 1 copy

Naval Research Laboratory
Technical Information Division
Code 2627
Washington, D. C. 20375 — 6 copies

Assistant Chief for Technology
Office of Naval Research
Code 200
Arlington, Virginia 22217 — 1 copy

Dr. A. L. Slafkosky
Scientific Advisor
Commandant of the Marine Corps
(Code RD-1)
Washington, D. C. 20380 — 1 copy

Office of Naval Research
Code 455
Arlington, Virginia 22217 — 1 copy

Office of Naval Research
Code 458
Arlington, Virginia 22217 — 1 copy

Naval Electronics Laboratory
Center
Advanced Software Technology
Division, Code 5200
San Diego, California 92152 — 1 copy

Mr. E. H. Gleissner
Naval Ship Research & Development
Center
Computation & Mathematics
Department
Bethesda, Maryland 20084 — 1 copy

Captain Grace M. Hopper
NAICOM/MIS Planning Branch
(OP-916D)
Office of Chief of Naval Operations
Washington, D. C. 20350 — 1 copy

Mr. Kin B. Thompson
Technical Director
Information Systems Division
(OP-91T)
Office of Chief of Naval Operations
Washington, D. C. 20350 — 1 copy